UNIVERSITA' DEGLI STUDI DI MILANO
Dipartimento di Tecnologie dell'Informazione
Polo Didattico e di Ricerca di Crema

LABORATORIO DI ALGORITMI

Appunti del corso di
Laboratorio di Algoritmi
del Professor Roberto Cordone

Vittoria Polimeni (670659)
Pietro Aiolfi (701927)

– Anno Accademico – 2006/2007 –

Indice

Introduzione

Questo corso di Laboratorio di Algoritmi vuole condurre lo studente lungo l'intero processo che permette di risolvere un problema con il calcolatore.

Il testo è strutturato in capitoli, ognuno dei quali è dedicato a un diverso problema. Si tratta di problemi giocattolo per contenere il tempo necessario a risolverli, ma quanto più possibile pratici. Dopo il problema, viene descritto in dettaglio il formato nel quale sono forniti i dati e quello nel quale si devono restituire i risultati. Quindi, una sezione è dedicata alla fase di *modellazione*, che individua gli oggetti matematici con cui rappresentare i dati e la soluzione del problema, motivando la scelta. Segue la fase di progetto dell'*algoritmo* che conduce dai dati alla soluzione e delle *strutture dati* che rendono l'algoritmo il più efficiente possibile. Infine, si ha la fase di stesura del *codice* che traduce l'algoritmo in un linguaggio di programmazione.

L'idea base del laboratorio è di *scomporre la stesura del codice in fasi*, per abituare lo studente a quel procedere metodico che è strettamente necessario ad affrontare programmi di grandi dimensioni scritti in collaborazione. Ogni fase occupa una distinta sezione e richiede allo studente di aggiungere al codice precedente poche semplici funzionalità che lo avvicinano alla realizzazione completa. Il più delle volte, la sezione fornisce le dichiarazioni delle procedure da implementare, con i relativi parametri di ingresso e uscita e con un corpo vuoto: allo studente il compito di definirle. Sono anche proposte alcune indicazioni sui costrutti del linguaggio C più adatti all'uopo. La sezione conclusiva riporta la soluzione dell'esercizio, con l'indicazione delle fasi del suo sviluppo. In appendice al testo, vengono richiamate alcune nozioni di base della sintassi del linguaggio C.

Il codice prodotto ad ogni fase deve essere sintatticamente corretto (compilare senza errori né avvertimenti, che spesso sono il segno ambiguo di errori insidiosi) e deve avere una relazione corretta fra ingresso e uscita, cioè deve essere eseguibile e fornire risultati verificabili. È chiaro che solo i risultati dell'ultima fase forniscono la soluzione del problema, ma le fasi intermedie devono fornire risultati parziali utili a individuare gli errori eventualmente commessi. Questo significa che la divisione del codice in fasi non è arbitraria,

ma risponde all'esigenza di tenerne lo sviluppo il più possibile sotto controllo.

Alcuni principi di stile Uno degli obiettivi che questo corso persegue è l'acquisizione di uno *stile* nella scrittura del codice, cioè l'aderenza a principi e regole. Lo stile si basa su motivazioni ragionate, per cui è in continua evoluzione e non va accettato a scatola chiusa, ma va corretto in base a motivazioni precise e non al semplice gusto.

Il codice deve essere *modulare* cioè *chiaramente diviso in procedure brevi che si scambiano informazioni esplicite*. Questo permette di:

- riutilizzare parti di codice la cui funzione è generale;
- visualizzare a colpo d'occhio ogni modulo (una regola di buon senso è che un modulo non superi mai un paio di schermate, ovvero una pagina a stampa);
- controllare le informazioni scambiate fra i moduli: *eviteremo sempre le variabili globali*, benché comode, perché consentono scambi nascosti di informazioni tra moduli, e quindi errori insidiosi e difficili da individuare;
- facilitare l'individuazione di
  - moduli scorretti, confrontando con un *debugger* il valore dei parametri prima e dopo l'esecuzione di un modulo con i valori corretti;
  - moduli inefficienti in tempo, sorvegliando con un *profiler* la percentuale del tempo di esecuzione trascorsa in ciascun modulo, così da capire dove sia più utile migliorare l'efficienza ;
  - moduli inefficienti in spazio, sorvegliando con il *monitor di sistema* la memoria allocata, così da individuare perdite dovute a una gestione errata.

Seguiremo il principio di *partire dal semplice per arrivare al complesso* e di *procedere dall'esterno verso l'interno*, cioè scrivere le procedure più semplici ed esterne (lettura dei dati, scrittura dei risultati, allocazioni e deallocazioni, ecc...) prima di quelle più complesse ed interne. Il vantaggio è di isolare il problema vero e proprio dalla sua interfaccia col mondo esterno (e via via il suo nucleo dai sottoproblemi meno specifici) e di poter verificare continuamente la correttezza del codice, sintattica (compilando) e semantica (verificando il legame ingresso-uscita su dati di test). Procedendo dall'esterno all'interno, siamo costretti a trattare le procedure non ancora implementate in termini di specifiche, cioè come "scatole nere" con morsetti di ingresso e

uscita. Questo rende anche più facile introdurre correzioni e adattamenti al codice.

Per tutte le soluzioni ai problemi del corso il programma principale (*main*) ha la stessa struttura:

1. interpretazione della linea di comando
2. caricamento dei dati e allocazione delle strutture per i dati e i risultati
3. eventuale allocazione di strutture ausiliarie
4. algoritmo
5. eventuale deallocazione delle strutture ausiliarie
6. restituzione dei risultati
7. deallocazione delle strutture per i dati e i risultati

caratterizzata dalla simmetria centrale rispetto al passo 4 (algoritmo). Inoltre, tutti gli svolgimenti procedono dalle procedure iniziali e finali a quella centrale (algoritmo), che è realizzata per ultima.

Lo stesso ordine (dall'esterno verso l'interno, anziché dall'inizio alla fine) si seguirà ad ogni livello del codice: si scriveranno apertura e chiusura di un file prima delle operazioni su di esso, inizio e termine di un ciclo o di un test prima delle operazioni in esso incluse, allocazione e deallocazione di una struttura dinamica prima delle operazioni che lo utilizzano, ecc...

La verifica continua della correttezza del codice richiede che in fase di sviluppo vengano eseguiti tutti i controlli possibili (entro limiti sensati) sulle fonti di errore (cioè sul successo nell'apertura di file, nella lettura di dati e stampa dei risultati, nell'allocazione e deallocazione di strutture dinamiche, ecc...). Verranno istituiti messaggi di errore significativi e univoci, per capire dove e quando si verifica l'errore (e quindi anche il perché). Inoltre, si utilizzeranno ovunque nomi simbolici al posto delle costanti numeriche per chiarirne il significato.

Riguardo i nomi delle variabili, è frequente nei corsi di ingegneria del software la raccomandazione di usare nomi lunghi e significativi. L'impostazione tendenzialmente matematica di questo corso ci porterà a trarre i nomi delle variabili e degli indici di scorrimento nei cicli direttamente dai nomi degli oggetti matematici cui corrispondono, e quindi a usare nomi brevi, ma in modo coerente, così che la corrispondenza puntuale del codice con la descrizione matematica del problema estenda la correttezza di quest'ultima al primo. Per far le barre, si può usare i package changebar o rcs.

Capitolo 1

Othello

1.1 Problema

Othello è un gioco da tavolo. Si gioca in due, su una scacchiera quadrata di 8 caselle per lato, usando pedine che hanno un lato bianco e l'altro nero. Il giocatore bianco posa sulla scacchiera le pedine con il lato bianco volto verso l'alto, quello nero con il lato nero. I giocatori si alternano posando una pedina alla volta su una casella vuota della scacchiera. Nel posarla, il giocatore deve catturare almeno una pedina di colore opposto: la nuova pedina e una vecchia dello stesso colore devono cioè racchiudere fra loro una sequenza orizzontale, verticale o diagonale di una o più pedine del colore opposto. Al termine della mossa, tutte le pedine catturate assumono il colore del giocatore che ha appena mosso. Con una sola mossa si possono catturare più linee di pedine contemporaneamente.

La figura 1.1 indica per una data posizione iniziale le mosse lecite per il giocatore bianco; la figura 1.2 indica la configurazione raggiunta dopo l'esecuzione di una di tali mosse.

Scrivete un programma per analizzare e commentare una sequenza di mosse di Othello.

1.1.1 Ingresso

L'ingresso consiste in una configurazione iniziale per la scacchiera, seguita da una lista di istruzioni. La configurazione della scacchiera consiste in 9 righe. Le prime 8 specificano lo stato corrente della scacchiera. Ognuna delle 8 righe contiene 8 caratteri, fra i seguenti:

- '-' indica una casella vuota
- 'B' indica una casella occupata da una pedina bianca

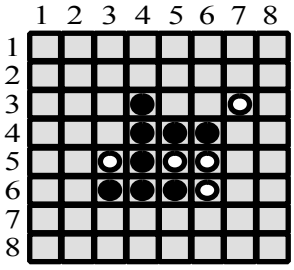

Figura 1.1: *Elenco delle mosse lecite per il giocatore Bianco:* $(2,3)$, $(3,3)$, $(3,5)$, $(6,2)$, $(7,3)$, $(7,4)$

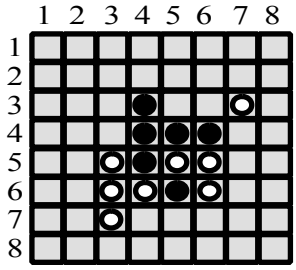

Figura 1.2: *Configurazione della scacchiera dopo che il giocatore Bianco ha eseguito la mossa* $(7,3)$

- 'N' indica una casella occupata da una pedina nera

La nona riga indica il giocatore corrente, e contiene una `B` o una `N`.

La lista delle istruzioni comincia con il loro numero. Segue un'istruzione per ogni riga, senza spazi bianchi, secondo il seguente formato:

`L` indica di elencare le mosse lecite per il giocatore corrente;

`M`xy (con x e y numeri compresi fra 1 e 8) indica che il giocatore corrente esegue una mossa, posando una pedina nella casella (x, y);

`Q` indica che le istruzioni da eseguire sono terminate.

Si assuma che i dati abbiano sempre un formato corretto. Le figure 1.3 e 1.4 riportano due esempi di ingresso.

1.1.2 Uscita

L'uscita deve contenere le reazioni dell'algoritmo alle istruzioni contenute nell'ingresso. Per ogni istruzione di tipo `L`, si deve visitare la scacchiera e stampare tutte le mosse lecite per il giocatore corrente, nel formato

```
(x,y)
```

dove `x` rappresenta la riga e `y` la colonna in cui il giocatore può posare una nuova pedina. Le mosse lecite devono essere stampate in ordine lessicografico, cioè:

1. se $x_1 < x_2$, tutte le mosse lecite nella riga x_1 vanno stampate prima di quelle nella riga x_2

```
--------
--------
--------
---BN---
---NB---
--------
--------
--------
B
4
L
M35
L
Q
```

Figura 1.3: *Primo esempio di ingresso*

```
BBBBN---
BBBN----
BBN-----
BN------
--------
--------
--------
--------
N
4
L
M25
L
Q
```

Figura 1.4: *Secondo esempio di ingresso*

2. se vi sono più mosse lecite nella riga x, vanno stampate in ordine di colonna y crescente

Le mosse lecite vanno stampate tutte su una sola riga e separate da uno spazio bianco. Se non vi sono mosse lecite perché il giocatore corrente non può catturare alcuna pedina, il programma deve stampare una riga contenente il messaggio

```
Nessuna mossa lecita
```

Per ogni istruzione di tipo `M`, se la mossa non è lecita, si passa la mano all'altro giocatore e gli si fa eseguire la mossa. Se anche per questo giocatore la mossa non è lecita, il programma stampa il messaggio

```
Mossa non lecita
```

Eseguita la mossa, si devono registrare le modifiche nella configurazione della scacchiera, cioè l'aggiunta della nuova pedina e il cambiamento di colore delle pedine catturate. Al termine della mossa, si deve stampare il numero di pedine di ciascun colore presenti sulla scacchiera, nel formato

```
Bianco - b Nero - n
```

dove `b` è il numero delle pedine bianche presenti sulla scacchiera e `n` è quello delle pedine nere. Dopo ogni mossa, il giocatore corrente viene sostituito dall'altro giocatore.

```
(3,5) (4,6) (5,3) (6,4)
Bianco -  4  Nero  - 1
(3,4) (3,6) (5,6)
--------
--------
----B---
---BB---
---NB---
--------
--------
--------
```

Figura 1.5: *Uscita corrispondente all'ingresso di Figura 1.3*

```
Nessuna mossa lecita
Bianco - 12  Nero  - 3
(3,5)
BBBBN---
BBBBB---
BBN-----
BN------
--------
--------
--------
--------
```

Figura 1.6: *Uscita corrispondente all'ingresso di Figura 1.4*

Per ogni istruzione di tipo `Q`, si stampi la configurazione finale della scacchiera, usando lo stesso formato dell'ingresso.

Le uscite corrispondenti ai due ingressi delle figure 1.3 e figure 1.4 sono riportati nelle figure 1.5 e figure 1.6.

1.2 Modello

Si tratta di rappresentare la scacchiera di Othello in un modo che consenta di gestire efficientemente le operazioni richieste dal problema, cioè:

1. valutare le mosse lecite, scorrendo la scacchiera in orizzontale, verticale e diagonale e verificando i colori delle pedine presenti nelle caselle.

2. aggiornare la scacchiera a seguito di una mossa, conservando l'indicazione del giocatore di turno e quella delle caselle vuote e piene e del colore delle pedine che le occupano.

Quindi serve poter accedere alle caselle in lettura (conoscere lo stato corrente) e in scrittura (per modificarlo), spostandosi fra caselle adiacenti in direzione orizzontale, verticale e diagonale.

Per questo esercizio la fase di modellazione è banale: conviene rappresentare la scacchiera con una matrice, perché questa struttura dati consente l'accesso diretto alle singole caselle, sia in lettura sia in scrittura, attraverso i loro indici di riga e colonna e tali indice consentono facilmente di spostarsi nelle direzioni richieste.

Faremo la scelta implementativa di conservare anche le istruzioni in una struttura dati opportuna, in modo da eseguirle nello stesso ordine in cui sono

fornite dal file d'ingresso. Poiché se ne conosce il numero sin dall'inizio, usare un vettore è una scelta adeguata.

1.3 Stesura del codice

1.3.1 Prima fase

Struttura generale del programma e definizione delle strutture dati

Il *main* segue la struttura generale descritta nell'introduzione. Nel codice `Othello1.c` le sue sottoprocedure sono già dichiarate con i parametri necessari per l'esecuzione, ma non ancora definite (il corpo è vuoto). Si può notare la mancanza delle operazioni di allocazione e deallocazione delle strutture ausiliarie, dovuta al fatto che l'esercizio è abbastanza semplice da non richiederne.

Tramite la direttiva `#define`, sono fornite alcune costanti simboliche per:

- la gestione degli errori (codici di uscita come `EXIT_SUCCESS`);
- i valore logici *vero* (`TRUE`, di valore 1, e `FALSE`, di valore 0).

Questi ultimi vengono sfruttati per definire il tipo enumerativo *enum _boolean*, che per semplicità viene ribattezzato *boolean* tramite il comando `typedef`.

Per semplificare la scrittura e usare nomi simbolicamente evocativi, il codice definisce con il comando `typedef` alcuni tipi ausiliari specifici di questo esercizio. Essi sono:

- *Scacchiera*: la scacchiera del gioco, rappresentata con una matrice bidimensionale di caratteri; poiché la sua dimensione è fissa, la scacchiera viene allocata staticamente;
- *Istruzione*: insieme delle proprietà che descrivono un'istruzione da eseguire, rappresentata con una `struct`;
- *VettoreIstruzioni*: insieme delle istruzioni da eseguire durante la partita, rappresentato con un vettore di *Istruzione*; verrà allocato dinamicamente perché la sua dimensione può cambiare ogni volta e si conosce solo dopo la lettura dei dati.

Inoltre, con la direttiva `#define` si forniscono le seguenti costanti simboliche:

- *LATO*: il lato della scacchiera;

- *LUNGHEZZA_ISTRUZIONE*: la lunghezza massima delle istruzioni;
- *VUOTO*, *BIANCO* e *NERO*: i possibili stati delle caselle (vuota, occupata da pedina bianca, occupata da pedina nera)

Si noti il passaggio dei parametri nella funzione *CaricaDati*: per convenzione, in tutto il corso i parametri di ingresso precedono i parametri di uscita. I parametri di uscita vengono passati per indirizzo perché la procedura li deve modificare. Quindi nelle chiamate sono preceduti da `&` e nelle definizioni divengono puntatori. Per sottolineare il fatto che i parametri di uscita non sono copie di *NumIstruz* e *I*, ma copie dei puntatori alle celle che li contengono, nella definizione essi assumono i nomi *pNumIstruzioni* e *pI*. Il parametro di uscita *S* fa eccezione: viene passato per valore perchè la procedura non modifica la matrice *S* (cioè il puntatore all'area di memoria occupata dalla scacchiera), ma solo il contenuto delle singole celle.

Il vettore *I* di tipo *Istruzione* viene passato per indirizzo e nella definizione diventa un puntatore *pI* a *VettoreIstruzioni* perché la procedura *CaricaDati* lo modifica allocando memoria e assegnandogliela. Si noti che un *VettoreIstruzioni* è implementato come puntatore a *Istruzione*, per cui *pI* è un puntatore doppio: l'uso della direttiva `typedef` nasconde questo dettaglio, che non ha nulla di concettuale, rendendo la scrittura più semplice e chiara.

1.3.2 Seconda fase

Lettura dei dati (scacchiera e istruzioni) e stampa della scacchiera

In questa fase, si devono definire le procedure più esterne e semplici:

- *LeggiLineaComando* interpreta la linea di comando. Questa procedura deve verificare che il numero di argomenti `argc` sia corretto; in caso affermativo, deve copiare il secondo argomento nel corrispondente parametro di uscita[1]; in caso negativo, deve restituire il messaggio d'errore: `Linea di comando errata` e interrompere l'esecuzione;
- *CaricaDati* gestisce il caricamento dei dati, cioè:
  - apre il file[2], segnalando eventuali errori con il messaggio: `Errore nell'apertura del file s` dove `s` sta per il nome del file di ingresso;
  - legge la scacchiera iniziale e il giocatore corrente;

[1] Suggerimento: `strcpy`
[2] Suggerimento: `fopen`, `fclose`

  - legge il numero delle istruzioni;
  - alloca il vettore delle istruzioni[3]
  - legge una per una le istruzioni stesse[4].

- *StampaScacchiera* stampa i risultati a video, scorrendo la matrice attraverso i suoi indici di riga e colonna[5].

1.3.3 Terza fase

Corpo dell'algoritmo e valutazione della scacchiera corrente

Questa fase richiede di strutturare la funzione *Othello*, che esegue le istruzioni sulla scacchiera e la tiene aggiornata. La funzione deve scorrere le istruzioni, interpretarle in base al primo carattere dell'istruzione stessa[6] ed eseguire le relative operazioni:

- `L`: scorre le caselle e valuta se è lecito al giocatore corrente posare una pedina in ciascuna casella attraverso la funzione

  ```
  boolean MossaLecita(int r,int c,char Giocatore,Scacchiera S);
  ```

  che dichiariamo ora e definiremo in seguito;

- `M`: valuta se la mossa è lecita per il giocatore corrente[7]; se lo è, la esegue attraverso la funzione

  ```
  void EsegueMossa(int r,int c,char Giocatore,Scacchiera S);
  ```

  che dichiariamo ora e definiremo in seguito; se non lo è, passa la mano all'altro giocatore[8] e se la mossa è diventata lecita, la esegue; eseguita la mossa, stampa il numero di pedine bianche e di pedine nere, ottenuto da un'opportuna funzione *ContaPedine* che va dichiarata e definita; se invece non c'è modo di eseguire la mossa, stampa il messaggio prescritto;

- `Q`: stampa a video la scacchiera e termina l'esecuzione delle istruzioni.

[3] Suggerimento: `calloc`
[4] Suggerimento: `printf` o `fgets`
[5] Suggerimento: `printf`
[6] Suggerimento: `switch`
[7] Suggerimento: per convertire in numeri interi gli indici di riga e colonna, che in *Istruzione* sono caratteri, si può usare l'aritmetica dei caratteri.
[8] Suggerimento: `if...then...else` oppure `(...?...:...)`

1.3.4 Quarta fase

Distinzione fra mosse lecite e non lecite

Questa fase richiede di definire la funzione *MossaLecita*, dichiarata nella fase precedente. Tale funzione:

- controlla che la casella di destinazione ricada nella scacchiera e sia libera;
- verifica che in una delle otto direzioni possibili (orizzontale verso destra e verso sinistra, verticale verso l'alto e verso il basso, diagonali), compaia una sequenza continua di pedine avversarie terminata da una pedina del giocatore corrente.

1.3.5 Quinta fase

Esecuzione delle mosse lecite

Questa fase richiede di definire la procedura *EsegueMossa*, la quale esegue una mossa che si assume lecita. Questa procedura ha struttura analoga a *MossaLecita*, ma modifica la scacchiera invece di leggerla soltanto.

1.3.6 Codice

Capitolo 2

Biblioteca

2.1 Problema

I mean your borrowers of books-those mutilators of collections, spoilers of the symmetry of shelves, and creators of odd volumes.
-Charles Lamb, Essays of Elia (1823) 'The Two Races of Men'

Parlo di quei prenditori in prestito di libri-quei mutilatori di collezioni, spogliatori della simmetria degli scaffali e creatori di volumi scompagnati. -Charles Lamb, Saggi di Elia (1823) 'Le Due Razze di Uomini'

Come Charles Lamb, i bibliotecari hanno qualche difficoltà con la gente che prende a prestito i libri: non li rimette dove dovrebbe. Per mantenere l'ordine, i libri restituiti vengono conservati al banco del prestito fino a che un bibliotecario non ha il tempo di rimetterli al posto giusto sugli scaffali. Anche per i bibliotecari, rimettere il libro giusto al posto giusto può essere una faccenda lunga. Ma siccome molte biblioteche oggi sono informatizzate, potete scrivere un programma per aiutarli.

Quando un utente prende o restituisce un libro, il computer ne conserva traccia. I libri restituiti vengono impilati sul banco: l'ultimo rimane così in cima. Periodicamente, i bibliotecari richiedono al programma la lista dei libri restituiti e l'indicazione di dove riporre ciascun libro, così da poterli rimettere al posto giusto sugli scaffali. Il programma deve restituire la lista dei libri ancora sugli scaffali e le istruzioni che i bibliotecari devono eseguire per riporli.

2.1.1 Ingresso

L'ingresso si compone di due file. Il primo elenca la dotazione della biblioteca, un libro per riga nel formato:

autore, "titolo"

I libri sono elencati in ordine alfabetico crescente rispetto all'autore e (in caso di autore identico oppure omonimo) rispetto al titolo. I libri sono disposti nello stesso ordine sullo scaffale della biblioteca (per semplicità, si suppone che vi sia uno scaffale solo). Si può assumere che la lunghezza di ciascuna riga (titolo e autore) non superi gli 80 caratteri e che tutti i libri abbiano autore o titolo diverso. Inizialmente, i libri sono tutti sullo scaffale. Il termine dell'inventario è indicato da una riga contenente solo la parola:

`FINE`

La Figura 2.1 riporta un esempio del primo file di ingresso.

```
Adams S., "Il principio di Dilbert"
Bertossi A., "Algoritmi e Strutture Dati"
Chaucer G., "I Racconti di Canterbury"
Hoeg P., "Il senso di Smilla per la neve"
Hofstadter D., "Godel, Escher, Bach"
Kernighan B. e Ritchie D., "Il linguaggio C"
King S., "L'ombra dello scorpione"
Pennac D., "La prosivendola"
Shakespeare W., "Le allegre comari di Windsor"
Yourcenar M., "L'opera al nero"
FINE
```

Figura 2.1: *Esempio di ingresso: file dell'inventario*

Il secondo file contiene le registrazioni dei prestiti e delle restituzioni. Ognuna appare su una singola riga, e rispetta il formato seguente:

`PRESTITO` *autore, "titolo"*

`RESTITUZIONE` *autore, "titolo"*

La lista termina con una riga che contiene solo la parola:

`FINE`

Questa va interpretata dal programma come una richiesta di assistenza nel riporre i libri restituiti. La Figura 2.2 riporta un esempio del primo file di ingresso.

```
PRESTITO Bertossi A., "Algoritmi e Strutture Dati"
PRESTITO Kernighan B. e Ritchie D., "Il linguaggio C"
PRESTITO Adams S., "Il principio di Dilbert"
RESTITUZIONE Kernighan B. e Ritchie D., "Il linguaggio C"
PRESTITO Shakespeare W., "Le allegre comari di Windsor"
PRESTITO Pennac D., "La prosivendola"
PRESTITO Yourcenar M., "L'opera al nero"
RESTITUZIONE Bertossi A., "Algoritmi e Strutture Dati"
RESTITUZIONE Shakespeare W., "Le allegre comari di Windsor"
FINE
```

Figura 2.2: *Esempio di ingresso: file delle istruzioni*

2.1.2 Uscita

Il programma deve restituire anzi tutto la lista dei libri ancora sullo scaffale, un libro per riga, inaugurata da una riga che contiene solo la parola chiave

`SCAFFALE`

e terminata da una riga che contiene solo la parola chiave:

`FINE`

Quindi, deve fornire una serie di istruzioni per i bibliotecari, inaugurata da una riga che contiene solo la parola chiave

`ISTRUZIONI`

cui segue un'istruzione per riga, nel formato:

`METTERE` *autore1, "titolo1"* `DOPO` *autore2, "titolo2"*

o, nel caso speciale di un libro che compare al primo posto nella collezione:

`METTERE` *autore, "titolo"* `PER PRIMO`

Questa serie di istruzioni termina con una riga che contiene solo la parola:

`FINE`

Infine, il programma deve restituire la lista dei libri che sono sullo scaffale al termine delle istruzioni di riordino, con lo stesso formato descritto in precedenza.

La Figura 2.3 riporta l'uscita corrispondente ai file di ingresso riportati nelle Figure 2.1 e 2.2.

```
SCAFFALE
Chaucer G., "I Racconti di Canterbury"
Hoeg P., "Il senso di Smilla per la neve"
Hofstadter D., "Godel, Escher, Bach"
King S., "L'ombra dello scorpione"
FINE
ISTRUZIONI
Mettere Shakespeare W., "Le allegre comari di Windsor" dopo King S., "L'ombra dello scorpione"
Mettere Bertossi A., "Algoritmi e Strutture Dati" per primo
Mettere Kernighan B. e Ritchie D., "Il linguaggio C" dopo Hofstadter D., "Godel, Escher, Bach"
FINE
SCAFFALE
Bertossi A., "Algoritmi e Strutture Dati"
Chaucer G., "I Racconti di Canterbury"
Hoeg P., "Il senso di Smilla per la neve"
Hofstadter D., "Godel, Escher, Bach"
Kernighan B. e Ritchie D., "Il linguaggio C"
King S., "L'ombra dello scorpione"
Shakespeare W., "Le allegre comari di Windsor"
FINE
```

Figura 2.3: *Esempio di uscita corrispondente all'ingresso delle Figure 2.1 e 2.2*

2.2 Modello

Le operazioni richieste sono:

- scorrere i libri sullo scaffale per stamparne l'elenco;
- tenere traccia dei libri prestati e dei libri resi;
- fornire informazioni sulla posizione dei libri da rimettere nella libreria;
- determinare la posizione corretta di un libro sullo scaffale in base all'ordine alfabetico.

Occorre quindi gestire tre insiemi di libri: quelli sullo scaffale, quelli in prestito e quelli restituiti. Il primo e il terzo insieme sono ordinati e, in particolare, il terzo consente solo l'accesso all'ultimo libro aggiunto all'insieme (*LIFO = Last In First Out*). Viene quindi spontaneo gestire almeno questi due insiemi come liste e per semplicità gestiremo così anche l'altro. Tutte le operazioni prima elencate verranno quindi realizzate come operazioni su liste.

2.3 Stesura del codice

2.3.1 Prima fase

Struttura generale del programma e definizione delle strutture dati

Il file `biblio1.c` fornisce tramite la direttiva `#define` alcune costanti simboliche per:

- la gestione degli errori (codici di uscita come `EXIT_SUCCESS`);
- la dimensione massima per le stringhe di caratteri (`RIGA`, di valore 80);
- i valore logici *vero* (`TRUE`, di valore 1, e `FALSE`, di valore 0).

Questi ultimi vengono sfruttati per definire il tipo enumerativo `enum _boolean`, che per semplicità viene ribattezzato *boolean* tramite il comando `typedef`.

Altri tipi ausiliari di dati specifici dell'esercizio sono:

- *AutoreTitolo*: è una stringa di caratteri, di lunghezza `RIGA`, che contiene autore e titolo di un libro
- *libro*: è una struttura contenente un *AutoreTitolo*

Infine, dato che raccoglieremo i libri in liste, per gestire queste liste viene fornita anche un'interfaccia che consente di eseguire operazioni su liste. Nomi e prototipi delle funzioni di tale libreria di interfaccia derivano dal capitolo 2 del testo di Algoritmi e Strutture Dati di A. Bertossi[1].

```
listalibri CreaListaLibri ();
void DistruggeListaLibri (listalibri *pL);
boolean listavuota (listalibri L);
poslibro primolista (listalibri L);
poslibro ultimolista (listalibri L);
poslibro succlista (poslibro p, listalibri L);
poslibro preclista (poslibro p, listalibri L);
boolean finelista (poslibro p, listalibri L);
char *leggilista (poslibro p, listalibri L);
listalibri scrivilista (char *AutoreTitolo, poslibro p, listalibri L);
void inslista (char *AutoreTitolo, poslibro p, listalibri L);
void canclista (poslibro *p, listalibri L);
poslibro trovaelemento (char *AutoreTitolo, listalibri L);
void StampaListaLibri (listalibri L);
```

In particolare, sono forniti i tipi ausiliari

[1] I nomi sono tutti autoesplicativi; l'unica precisazione necessaria è che la funzione *inslista* inserisce il suo primo argomento nella posizione che precede il secondo argomento.

- *listalibri* per descrivere una lista
- *poslibro* per descrivere un indice di scorrimento lungo una lista

Poiché la lista viene realizzata come doppia lista a puntatori circolare e con sentinella, entrambi i tipi sono definiti come puntatori a *libro* e la struttura *libro* contiene anche un puntatore al libro successivo (*next*) e uno al libro precedente (*prev*), ovviamente di valore `NULL` per libri che non appartengono a nessuna lista.

Nel *main* sono definite tre variabili di tipo *listalibri*:

- *Scaffale*, che contiene i libri presenti nella libreria;
- *Prestati*, che contiene i libri in prestito;
- *Resi*, che contiene i libri resi e non riposti.

La struttura del main è la consueta, salvo che si eseguono due algoritmi anziché uno (l'esecuzione dei movimenti dei libri e il reinserimento dei libri resi a fine giornata) e ciascuno di loro è seguito dalla stampa dei suoi risultati.

2.3.2 Seconda fase

Interpretazione della linea di comando e caricamento della dotazione della biblioteca

Questa fase richiede di definire le seguenti funzioni:

- *LeggeLineaComando* che interpreta la linea di comando: somiglia a quella definita nell'esercizio precedente, ma gli argomenti in ingresso sono due: il file dei libri disponibili e quello dei movimenti;
- *CaricaBiblioteca* che legge dal primo file di ingresso una riga alla volta[2] e lo stampa a video finché non legge la terminazione `FINE`[3].

2.3.3 Terza fase

Costruzione della lista Scaffale *e principio della libreria per gestire liste*

[2]Suggerimento: `fgets`, e `strlen` per eliminare l'a capo finale
[3]Suggerimento: `strcmp`

Questa fase richiede di modificare la procedura *CaricaBiblioteca* in modo che le stringhe *AutoreTitolo* lette via via non vengano più stampate a video, ma inserite nella lista *Scaffale*[4].

Questa fase richiede inoltre di definire le seguenti procedure per la gestione di liste:

- *CreaListaLibri*: crea una lista di libri vuota, cioè contenente la sola sentinella[5]; se la memoria è insufficiente, esce con un messaggio di errore.
- *DistruggeListaLibri*: scorre una lista eliminandone gli elementi uno ad uno finché la lista non è vuota[6]; quindi, dealloca la sentinella.
- *StampaListaLibri*: scorre una lista stampando per ogni libro l'autore e il titolo[7].

2.3.4 Quarta fase

Conclusione della libreria per gestire liste

Questa fase richiede di definire le funzioni rimanenti della libreria:

- *primolista*, che restituisce il primo elemento della lista;
- *listavuota*, che restituisce `TRUE` qualora la lista sia vuota, `FALSE` altrimenti;
- *canclista*, che cancella l'elemento indicato dalla lista e aggiorna la posizione spostandola sul successivo;
- *finelista* che restituisce `TRUE` qualora la posizione corrente coincida con quella finale della lista, `FALSE` altrimenti.
- *leggilista* che restituisce il titolo e l'autore del libro in posizione p nella lista L.

[4]Suggerimento: occorrono le funzioni *inslista* e *ultimolista* (e *succlista*, dato che *inslista* inserisce il primo argomento nella posizione che precede il secondo argomento anziché in quella successiva).

[5]Suggerimento: `malloc`

[6]Suggerimento: *primolista*, *listavuota*, *canclista*

[7]Suggerimento: *primolista*, *finelista*, *succlista*, *leggilista*.

2.3.5 Quinta fase

Esecuzione di prestiti e rese dal punto di vista delle liste dei libri prestati e resi, aggiornando anche la lista dei libri disponibili

Questa fase richiede di realizzare l'algoritmo vero e proprio, gestendo l'esecuzione di prestiti e rese, cioè modificando le corrispondenti liste. Va definita la funzione *EsegueMovimenti* che:

- in caso di prestito:
  1. cerca il libro sullo scaffale, chiamando un'apposita funzione *trovaelemento* (da dichiarare solamente), e lo cancella
  2. inserisce il nuovo libro in coda alla lista dei prestiti
- in caso di resa:
  1. cerca il libro nella lista dei prestiti con la funzione *trovaelemento* e lo cancella
  2. inserisce il libro in cima alla lista dei resi
- in fase di riordino
  1. cerca il libro in cima alla lista dei resi e lo cancella
  2. lo aggiunge nella posizione corretta sullo scaffale
  3. completa l'esecuzione dei prestiti e dei resi aggiornando anche la lista dei libri disponibili

2.3.6 Sesta fase

Reinserimento dei libri resi e definizione di findelemento

Questa fase richiede di definire le funzioni:

- *trovaelemento* che scorre una lista di libri alla ricerca di un determinato elemento[8].
- *RiponeLibriRestituiti*, che cerca nella lista *Scaffale* la posizione corretta in base all'ordine alfabetico[9] per ciascun libro della lista *Resi* e lo inserisce in essa.

2.3.7 Codice

[8]Suggerimento: *primolista*, *finelista*, *succlista*, *leggilista*, `strcmp`
[9]Suggerimento: *primolista*, *finelista*, *succlista*, *leggilista*, `strcmp`

Capitolo 3

Dynasty

3.1 Problema

La tradizione delle monarchie occidentali stabilisce la linea di successione al trono ordinando gli individui appartenenti alla famiglia reale in base alle seguenti regole:

1. ogni individuo è immediatamente seguito dal figlio primogenito;
2. se l'individuo non ha figli, è seguito dal fratello immediatamente più giovane;
3. se l'individuo non ha neppure fratelli minori, è seguito dal fratello immediatamente più giovane del più vicino antenato (padre, nonno, ecc...) che abbia discendenti o fratelli minori;

Quando un sovrano muore, o abdica, sale al trono il primo successore vivente.

Si vuole scrivere un programma che riceva in ingresso un elenco di personaggi storici membri di una famiglia, con le relative date di nascita e di morte e le relazioni padre-figlio, e fornisca in uscita la successione dei sovrani e l'estensione temporale del loro regno. Il programma dovrà quindi stampare un elenco di tutti i personaggi (in ordine qualsiasi), indicando per ognuno il numero di discendenti riportati nell'elenco e il numero di quelli ascesi al trono. Fra i discendenti di un individuo non va contato l'individuo stesso.

Assunzioni semplificative

1. Si ignora la successione in linea femminile.
2. Ogni personaggio ha al più due figli.

3. Ogni personaggio ha un nome diverso, di lunghezza non superiore a 30 caratteri.

4. La data di inizio del regno per il capostipite coincide con la data di nascita.

5. Ogni regno termina in un anno strettamente successivo a quello in cui comincia.

3.1.1 Ingresso

L'ingresso si compone di un elenco di personaggi storici. La prima riga riporta i dati del capostipite e segue il formato:

nome `[`*nascita* `-` *morte*`]` `capostipite`

Le righe seguenti riportano ciascuna i dati di un personaggio, secondo il formato:

nome `[`*nascita* `-` *morte*`]` `da` *nome padre*

L'elenco termina con una riga contenente solo la parola:

```
FINE
```

Si assuma che nessun personaggio preceda il proprio padre.

La Figura 3.1 riporta come esempio di ingresso parte dell'albero genealogico della famiglia Medici (benché non famiglia reale, a rigore).

3.1.2 Uscita

Il programma deve restituire anzi tutto la lista dei sovrani, uno per riga in ordine cronologico di ascesa al trono, con le date di ascesa al trono e di morte, nel formato:

nome `[`*ascesa* `-` *morte*`]`

terminata da una riga che contiene solo la parola chiave:

```
FINE
```

Quindi, deve fornire l'elenco dei personaggi storici, indicando per ognuno il numero dei discendenti e quello dei discendenti ascesi al trono, secondo il formato:

nome`:` *numero discendenti* `di cui` *numero* `sovrani`

```
Giovanni_di_Bicci [1360-1429] capostipite
Cosimo_il_Vecchio [1389-1464] da Giovanni_di_Bicci
Lorenzo_il_Vecchio [1395-1440] da Giovanni_di_Bicci
Piero_il_Gottoso [1416-1469] da Cosimo_il_Vecchio
Pier_Francesco [1430-1476] da Lorenzo_il_Vecchio
Lorenzo_il_Magnifico [1449-1492] da Piero_il_Gottoso
Giuliano [1453-1478] da Piero_il_Gottoso
Lorenzo_il_Popolano [1463-1503] da Pier_Francesco
Giovanni_il_Popolano [1467-1498] da Pier_Francesco
Piero_lo_Sfortunato [1472-1503] da Lorenzo_il_Magnifico
Giovanni [1475-1521] da Lorenzo_il_Magnifico
Giulio [1478-1534] da Giuliano
Pier_Francesco_II [1487-1525] da Lorenzo_il_Popolano
Giovanni_dalle_Bande_Nere [1498-1526] da Giovanni_il_Popolano
Lorenzo [1492-1519] da Piero_lo_Sfortunato
Alessandro [1511-1537] da Giulio
Lorenzino [1514-1548] da Pier_Francesco_II
Cosimo_I [1519-1574] da Giovanni_dalle_Bande_Nere
FINE
```

Figura 3.1: *Esempio di ingresso*

Anche questo elenco termina con una riga che contiene solo la parola chiave:

```
FINE
```

La Figura 3.2 riporta l'uscita corrispondente all'ingresso della Figura 3.1.

3.2 Modello

Il problema richiede di rappresentare la relazione di discendenza (diretta e indiretta) fra personaggi. Questa relazione induce sui personaggi una struttura ad albero. Le regole tradizionali di successione al trono corrispondono alla visita in profondità di tale albero. D'altra parte, i conteggi sul numero di discendenti (o di discendenti aventi opportune proprietà) richiedono altrettante visite.

Grazie alle ipotesi semplificative, l'albero in questione è un albero binario[1]. Un albero si compone di *nodi*, che contengono informazioni e *archi*, che

[1] È giusto osservare che, anche se ogni personaggio avesse un generico numero di figli, si potrebbe rappresentare la famiglia con un albero binario nel quale il ramo sinistro uscente da ogni nodo individuasse il primogenito e il destro individuasse il fratello immediatamente più giovane. Ovviamente, cambierebbero gli algoritmi di visita, con qualche complicazione.

```
Giovanni_di_Bicci [1360-1429]
Cosimo_il_Vecchio [1429-1464]
Piero_il_Gottoso [1464_1469]
Lorenzo_il_Magnifico [1469-1492]
Piero_lo_sfortunato [1492-1503]
Lorenzo [1503-1519]
Giovanni [1519-1521]
Giulio [1521-1534]
Alessandro [1534-1537]
Lorenzino [1537-1548]
Cosimo_I [1548-1574]
FINE
Lorenzo: 0 di cui 0 sovrani
Piero_lo_Sfortunato: 1 di cui 1 sovrani
Giovanni: 0 di cui 0 sovrani
Lorenzo_il_Magnifico: 3 di cui 3 sovrani
Alessandro: 0 di cui 0 sovrani
Giulio: 1 di cui 1 sovrani
Giuliano: 2 di cui 2 sovrani
Piero_il_Gottoso: 7 di cui 6 sovrani
Cosimo_il_Vecchio: 8 di cui 7 sovrani
Lorenzino: 0 di cui 0 sovrani
Pier_Francesco_II: 1 di cui 1 sovrani
Lorenzo_il_Popolano: 2 di cui 1 sovrani
Cosimo_I: 0 di cui 0 sovrani
Giovanni_dalle_Bande_Nere: 1 di cui 1 sovrani
Giovanni_il_Popolano: 2 di cui 1 sovrani
Pier_Francesco: 6 di cui 2 sovrani
Lorenzo_il_Vecchio: 7 di cui 2 sovrani
Giovanni_di_Bicci: 17 di cui 10 sovrani
FINE
```

Figura 3.2: *Esempio di uscita corrispondente all'ingresso di Figura 3.1*

collegano fra loro gerarchicamente due nodi (un *padre* e un *figlio*). In un albero binario, ogni nodo ha al più due figli, definiti *sinistro* e *destro*. Se non ne ha, si definisce *foglia*. Infine, esiste uno e un solo nodo nell'albero che non è figlio di nessun'altro nodo, e viene definito *radice*.

3.3 Stesura del codice

3.3.1 Prima fase

Struttura generale del programma e definizione delle strutture dati

Il *main* segue la struttura generale descritta nell'introduzione. Nel codice `Dynasty1.c` le sue sottoprocedure sono già dichiarate con i parametri necessari per l'esecuzione, ma non ancora definite (il corpo è vuoto).

Oltre alle consuete costanti simboliche per definire il tipo `boolean` e per gestire gli errori (cui si è aggiunta una costante `EXIT_INCONSISTENCY` per gestire il caso di struttura dati incoerente), la direttiva `#define` fornisce le seguenti costanti specifiche dell'esercizio:

- *LUNGHEZZA_NOME* (pari a 30) per limitare la lunghezza dei nomi dei personaggi;
- *LUNGHEZZA_RIGA* (pari a due nomi più 24, per limitare la lunghezza della riga di lettura così da poter acquisire il nome di ogni personaggio insieme a quello del padre nel formato descritto sopra;
- *DATAFASULLA* (pari a -1) per indicare date non ancora definite.

Per rappresentare i personaggi storici è definito il tipo ausiliario

- *Personaggio*, che fornisce il nome e le date di nascita e morte (che si ricavano dai dati), nonché la data di eventuale ascesa al trono e il numero di discendenti e discendenti sovrani (che sono risultati dell'elaborazione).

Dato che raccoglieremo i personaggi in un albero binario, viene fornita una libreria di interfaccia per gestire tale albero. Nomi e prototipi delle funzioni di tale interfaccia derivano dal capitolo 4 del testo di Algoritmi e Strutture Dati di A. Bertossi[2].

```
binalbero creabinalbero ();
boolean binalberovuoto (binalbero T);
void insbinradice (nodo u, binalbero *pT);
nodo binradice (binalbero T);
```

[2]Con qualche piccola licenza.

```
nodo binpadre (nodo u, binalbero T);
nodo figliosinistro (nodo u, binalbero T);
nodo figliodestro (nodo u, binalbero T);
boolean sinistrovuoto (nodo u, binalbero T);
boolean destrovuoto (nodo u, binalbero T);
void inssottobinalberosinistro (binalbero ST, nodo u, binalbero *pT);
void inssottobinalberodestro (binalbero ST, nodo u, binalbero *pT);
void legginodo (nodo n, binalbero T, char *nome, int *pnascita,
                int *pmorte, int *pascesa, int *pdisc, int *pdisc_sovr);
void scrivinodo (nodo n, char *nome, int nascita, int morte, int ascesa,
                 int disc, int disc_sovr);
void cancsottobinalbero (nodo u, binalbero *pT);
```

In particolare, sono forniti i tipi ausiliari

- *binalbero* per descrivere alberi binari,
- *nodo* per descrivere nodi di albero binari.

Entrambi sono definiti come puntatori a *Personaggio* perché l'albero viene realizzato attraverso puntatori. Quindi, le costanti simboliche `EMPTY_TREE` che rappresenta l'albero vuoto e `NO_NODE` che rappresenta un nodo inesistente, vengono definite come `NULL`.

Alla struttura *Personaggio* vengono quindi aggiunti i tre campi *padre*, che è un nodo, *figlio_sx* e *figlio_dx*, che sono alberi binari, in modo da realizzare la struttura gerarchica ad albero.

3.3.2 Seconda fase

Interpretazione della linea di comando, creazione semplificata dell'albero genealogico

In questa fase si deve procedere a definire le seguenti funzioni:

- *LeggeIstruzione*, che acquisisce dalla linea di comando il nome del file che contiene le relazioni di discendenza;
- *CaricaDiscendenze*, che
  - apre il file di ingresso[3];
  - crea un albero vuoto e gli assegna una radice;
  - legge il nome e le date del capostipite e le scrive nella radice[4];

[3]Suggerimento: `fopen`, `fclose`

[4]Suggerimento: *scrivinodo*

– legge i nomi dei personaggi e le loro relazioni di discendenza[5];
– si ferma appena legge la parola chiave `FINE` anziché il nome di un personaggio;
– inserisce personaggi e relazioni nella struttura ad albero;

Per semplicità, cominceremo creando per ogni personaggio un albero binario vuoto, assegnandogli una radice, scrivendovi i dati del personaggio e appendendo il nuovo albero come sottoalbero sinistro al nodo precedente. La funzione *CaricaDiscendenze* si può quindi costruire con le funzioni di libreria (che definiremo in seguito):

- *creabinalbero*, che crea un albero vuoto;
- *insradice*, che alloca un nodo[6] (assegnando valori di *default* ai suoi campi) e lo inserisce come radice in un albero vuoto
- *scrivinodo*, che assegna a un nodo il nome e le date fornite come argomento
- *inssottobinalberosinistro*, che inserisce un sottoalbero dato come sottoalbero sinistro di un albero dato in un nodo dato

3.3.3 Terza fase

Prima fase della definizione della libreria per gestire alberi binari

Questa fase prevede la definizione delle funzioni già precedentemente usate:

- *creabinalbero*, che crea un albero vuoto;
- *insbinradice*, che alloca un nodo[7] (assegnando valori di *default* ai suoi campi) e lo inserisce come radice in un albero vuoto
- *scrivinodo*, che assegna a un nodo il nome e le date fornite come argomento
- *inssottobinalberosinistro*, che inserisce un sottoalbero dato come sottoalbero sinistro di un albero dato in un nodo dato

nonché delle seguenti:

[5]Suggerimento: `fscanf`
[6]Suggerimento: `malloc`
[7]Suggerimento: `malloc`

- *binalberovuoto*, che verifica se un albero è vuoto
- *figliodestro*, che restituisce il nodo figlio destro del nodo dato nell'albero dato
- *figliosinistro*: restituisce il nodo figlio sinistro del nodo dato nell'albero dato
- *binpadre*: restituisce il nodo padre del nodo dato nell'albero dato
- *sinistrovuoto*: indica se il sottoalbero sinistro del nodo dato nell'albero dato è vuoto
- *destrovuoto*: : indica se il sottoalbero destro del nodo dato nell'albero dato è vuoto
- *binradice*: restituisce il nodo radice dell'albero indicato
- *legginodo*, che restituisce i dati associati al nodo dato dell'albero dato

3.3.4 Quarta fase

Seconda fase della definizione della libreria per gestire alberi binari e correzione della procedura di costruzione dell'albero

Questa fase conclude la definizione della libreria con la funzione

- *cancbinalbero*, che dealloca tutti i nodi dell'albero dato sino a trasformarlo in un albero vuoto (ovviamente, ciò avviene visitandolo in "post-ordine")

e definisce le funzioni specifiche dell'applicazione agli alberi genealogici:

- *CercaPersonaggio*, che scorre un albero binario dato sino a rintracciare il nodo corrispondente al nome desiderato, procedendo in pre-ordine (ovviamente, qualsiasi altro ordine sarebbe corretto); se il personaggio non appartiene all'albero, restituisce `NO_NODE`;
- *StampaAlbero*, che stampa i personaggi dell'albero binario dato in "in-ordine".

Inoltre, potendo ora disporre della procedura che determina entro l'albero genealogico il padre di ogni personaggio a partire dal suo nome (purché nel file di ingresso il padre venga riportato sempre prima del figlio), in questa fase si deve correggere la procedura *CaricaDiscendenza* determinando per

ogni personaggio il nodo corrispondente al padre e appendendogli l'albero associato al figlio come sottoalbero sinistro se non è ancora presente alcun figlio, come sottoalbero destro se ne è già presente uno. Si assuma che non possano essere presenti più di due figli.

Nota bene: In tutte le procedure ricorsive su alberi binari, il caso base della ricorsion è quello di *albero vuoto*; un albero ridotto alla sola radice è un albero semplice, ma già inutilmente complicato rispetto all'albero vuoto.

3.3.5 Quinta fase

Calcolo delle date di ascesa al trono e stampa della lista dei sovrani

Questa fase definisce le funzioni

- *CalcolaSovrani*, che calcola le date di ascesa al trono; a tale scopo, effettua una visita in pre-ordine dell'albero, trasmettendo di chiamata in chiamata attraverso l'argomento *pdata* la data di morte dell'ultimo sovrano:
  - se il personaggio corrente muore prima di tale data, non può salire al trono (e la sua data di ascesa al trono rimane pari a `DATA_FASULLA`)
  - se il personaggio corrente è ancora vivo alla morte del precedente sovrano, sale al trono diventando il nuovo sovrano, il che comporta l'aggiornamento di *pdata* (ed è per questo che l'argomento viene passato per indirizzo) e del campo relativo alla data di ascesa al trono;
- *StampaSovrani*, che stampa la lista dei sovrani nell'ordine corrispondente alla loro salita al trono; si tratta di ripercorrere l'identica visita, stampando il nome del sovrano se la data di ascesa al trono è significativa, nulla se non lo è;

3.3.6 Sesta fase

Calcolo e stampa del numero di discendenti e di discendenti sovrani

Questa fase, analogamente alla precedente, definisce le funzioni

- *CalcolaDiscendenti*, che calcola il numero di discendenti e di discendenti sovrani di ogni personaggio, attraverso una procedura ricorsiva basata sulla semplice constatazione che il numero di discendenti di un nodo è 0 se il nodo è vuoto; altrimenti è dato dal numero di discendenti del

figlio sinistro più quello del figlio destro, più 1 per il figlio sinistro se esso non è vuoto e 1 per il figlio destro se esso non è vuoto

- *StampaDiscendenti*, che stampa l'elenco dei discendenti di un sovrano evidenziando anche quanti discendenti sono a loro volta diventati sovrani.

3.3.7 Codice

Capitolo 4

Entity

4.1 Problema

Un programma di Intelligenza Artificiale ha il compito di riconoscere e manipolare concetti. In particolare, deve saper riconoscere definizioni diverse che si riferiscono alla stessa entità. Solo in questo modo, infatti, può interagire correttamente con chi ne parla. Un modo per dotare il programma di questa capacità è fornirgli enunciati che affermino l'identità fra definizioni diverse della stessa entità (ad esempio, enunciati come "A = B" o "B = C"). Poiché l'identità è una relazione di equivalenza, gode delle proprietà simmetrica e transitiva. Di conseguenza, un sistema di enunciati di identità può contenere ridondanze, come "B = A" e "A = C".

Scrivete un programma che riceve in ingresso una sequenza di definizioni e una sequenza di enunciati di identità, identifica un insieme minimale di enunciati di identità e stampa l'elenco di quelli eliminati perché ridondanti. Infine, il programma deve stampare per ogni entità distinta una definizione completa, cioé l'elenco di tutte le sue definizioni, partendo da quella "fondamentale", che definiamo per convenzione come la più breve.

4.1.1 Ingresso

L'ingresso si compone di due file. Il primo si apre con una riga che contiene il numero di definizioni, seguita dalle definizioni stesse, una per riga. Si assuma che le definizioni non superino mai gli 80 caratteri. Il secondo file si apre con una riga che contiene il numero di enunciati di identità, seguito dagli enunciati stessi, uno per riga, nel formato

definizione1=*definizione2*

```
11
PRIMO IMPERATORE DI FRANCIA
AMANTE DI LADY HAMILTON
ALESSANDRO I
VINCITORE DI LIPSIA
ZAR DI TUTTE LE RUSSIE DAL 1801 AL 1825
VINCITORE DI TRAFALGAR
NAPOLEONE
AMANTE DI JOSEPHINE BOHARNAIS
LORD NELSON
PRIMO MINISTRO INGLESE DAL 1804 AL 1806
WILLIAM PITT IL GIOVANE
```

Figura 4.1: *Esempio di ingresso: file delle definizioni*

```
9
PRIMO IMPERATORE DI FRANCIA=VINCITORE DI LIPSIA
VINCITORE DI LIPSIA=AMANTE DI JOSEPHINE BOHARNAIS
ZAR DI TUTTE LE RUSSIE DAL 1801 AL 1825=ALESSANDRO I
VINCITORE DI TRAFALGAR=LORD NELSON
NAPOLEONE=AMANTE DI JOSEPHINE BOHARNAIS
AMANTE DI JOSEPHINE BOHARNAIS=PRIMO IMPERATORE DI FRANCIA
NAPOLEONE=NAPOLEONE
LORD NELSON=VINCITORE DI TRAFALGAR
LORD NELSON=AMANTE DI LADY HAMILTON
```

Figura 4.2: *Esempio di ingresso: file degli enunciati di identità*

dove *definizione1* e *definizione2* sono definizioni tratte dal primo file e sono separate solo dal carattere `=` senza alcuno spazio intermedio.

Un esempio di ingresso è riportato nelle Figure 4.1 e 4.2.

4.1.2 Uscita

Il programma stampa una riga nel formato

```
Enunciati ridondanti
```

seguita dagli enunciati di identità ridondanti, uno per riga. In caso non vi siano ridondanze, il programma stampa

```
Enunciati ridondanti:  nessuno
```

Quindi, il programma stampa il numero di entità distinte, in una riga dal formato

```
Enunciati ridondanti
AMANTE DI JOSEPHINE BOHARNAIS=PRIMO IMPERATORE DI FRANCIA
NAPOLEONE=NAPOLEONE
LORD NELSON=VINCITORE DI TRAFALGAR
5 entità distinte
Entità ALESSANDRO I
ZAR DI TUTTE LE RUSSIE DAL 1801 AL 1825
Entità NAPOLEONE
       PRIMO IMPERATORE DI FRANCIA
       VINCITORE DI LIPSIA
       AMANTE DI JOSEPHINE BOHARNAIS
Entità LORD NELSON
       AMANTE DI LADY HAMILTON
       VINCITORE DI TRAFALGAR
Entità PRIMO MINISTRO INGLESE DAL 1804 AL 1806
Entità WILLIAM PITT IL GIOVANE
```

Figura 4.3: *Esempio di uscita corrispondente all'ingresso delle Figure 4.1 e 4.2*

numero `entità distinte`

dove *numero* è il numero delle entità stesse, seguita dalla definizione completa di ogni entità. Per ciascuna si riportano le definizioni, una per riga, con la definizione fondamentale nella prima riga, preceduta dalla parola chiave `Entità`, con le definizioni indentate in modo da incolonnarsi sotto quella fondamentale, come segue:

`Entità` *Descrizione fondamentale*

Descrizione1

Descrizione2

...

La Figura 4.3 riporta un esempio di uscita corrispondente all'ingresso delle Figure 4.1 e 4.2.

4.2 Modello

Dato l'insieme delle definizioni, ogni entità è una classe di equivalenza rispetto alla relazione di identità, cioè un sottoinsieme di definizioni disgiunto da ogni altro sottoinsieme. Per costruire tali sottoinsiemi basta partire con

un sottoinsieme per ogni definizione, come se ognuna corrispondesse a una diversa entità, e ogni volta che un enunciato stabilisce l'identità di due definizioni A e B, individuare a quali sottoinsiemi (entità) appartengono e fonderli se sono distinti.

Di conseguenza, quello che ci serve è rappresentare una partizione di un insieme dato in sottoinsiemi disgiunti. Le operazioni fondamentali di cui si deve poter disporre sono:

- determinare a che sottoinsieme appartiene ogni elemento
- fondere due sottoinsiemi

Queste operazioni sono realizzate in modo molto efficiente con i *merge-find set*.

4.3 Codice

4.3.1 Prima fase

Struttura generale del programma e definizione delle strutture dati

Il codice `Entity1.c` fornisce le consuete costanti simboliche per definire il tipo `boolean`, gestire gli errori e indicare la lunghezza massima del nome di un file (*LUNGHEZZA*, pari a 255). In aggiunta, viene definita una costante specifica dell'esercizio:

- *DEFINIZIONE* (pari a 80) per limitare la lunghezza delle definizioni.

Sono inoltre definiti tramite `typedef` i seguenti tipi di dato:

- *Definizione* per rappresentare una definizione: è un semplice vettore di caratteri allocato staticamente;
- *VettoreDefinizioni* per rappresentare l'insieme delle definizioni: è un vettore di definizioni allocato dinamicamente;
- *Enunciato* per rappresentare una coppia di definizioni di cui si afferma l'identità: contiene due indici numerici che individuano una definizione entro un vettore e un valore logico che indica se l'enunciato è ridondante o no;
- *VettoreEnunciati* per rappresentare l'insieme degli enunciati: è un vettore di enunciati allocato dinamicamente.

Bisogna poi definire strutture per rappresentare le entità. Come si è detto, esse costituiscono una collezione di sottoinsiemi disgiunti di definizioni, rappresentata tramite *merge-find set*. Viene fornita una libreria di interfaccia per gestire tale struttura. Nomi e prototipi delle funzioni di tale interfaccia derivano dalla Sezione 5.5 del testo di Algoritmi e Strutture Dati di A. Bertossi[1].

```
mfset creamfset (insieme A, int CardA);
componente trova (elemento x, mfset S);
mfset fondi (componente c1, componente c2, mfset S);
```

In particolare, sono forniti i tipi ausiliari:

- *insieme*, che descrive l'insieme delle definizioni, e quindi coincide con il tipo *VettoreDefinizioni* à definito;
- *elemento*, che descrive un elemento all'interno di tale insieme ed è definito come un indice intero;
- *componente*, che descrive un sottoinsieme di tale insieme ed è definito come un indice intero;
- *nodo*, che contiene le strutture necessarie a gestire ciascun elemento all'interno del *merge-find set*: si tratta dell'indice dell'elemento padre, cioè per ogni definizione l'indice della definizione alla quale essa viene ricondotta (su su fino alla definizione radice dell'entità), e di un indice di rango che specifica quanti livelli di definizioni si possono trovare sotto quello corrente;
- *mfset*, che rappresenta l'intera collezione dei sottoinsiemi, ed è definito come un vettore di *nodo* allocato dinamicamente.

Il *main* contiene la consueta struttura generale del programma, con una struttura ausiliaria (il *mfset* S, appunto), che viene allocata e deallocata.

4.3.2 Seconda fase

Interpretazione della linea di comando e lettura delle definizioni

In questa fase si deve procedere a definire le seguenti funzioni:

- *LeggeIstruzioni*, che acquisisce dalla linea di comando il nome del file che contiene le descrizioni e di quello che contiene gli enunciati di identità;

[1] Con qualche piccola correzione.

- *CaricaDefinizioni*, che
  - apre il file delle definizioni[2];
  - legge il numero delle definizioni;
  - alloca il vettore delle definizioni (ovviamente passato per indirizzo);
  - legge riga per riga le definizioni[3];

4.3.3 Terza fase

Lettura degli enunciati e identificazione delle due definizioni che compongono ciascun enunciato

In questa fase si deve definire la funzione *CaricaEnunciati*, che

- apre il file degli enunciati;
- legge il numero delle enunciati di identità;
- alloca il vettore degli enunciati (ovviamente passato per indirizzo);
- legge riga per riga gli enunciati[4];
- cerca le due definizioni nel relativo vettore e ne individua gli indici.

4.3.4 Quarta fase

Realizzazione della libreria per gestire i merge-find set

In questa fase, si devono realizzare le funzioni di base per la gestione dei *merge-find* set, vale a dire:

- *creamfset*, che costruisce la struttura stessa, sulla base dell'insieme da partizionare in sottoinsiemi disgiunti e della sua cardinalità;
- *trova*, che determina la componente di cui fa parte un elemento dato di un *merge-find set* dato;

[2]Suggerimento: `fopen`, `fclose`

[3]Suggerimento: è possibile sia leggere le righe con `fgets` ed eliminare a posteriori l'a capo finale sia leggerle con `fscanf` impiegando la stringa di formato formato `[^\n]` che riconosce qualsiasi stringa di caratteri interrompendola alla prima occorrenza di `\n`.

[4]Suggerimento: è possibile distinguere la prima e la seconda definizione con `fscanf` impiegando la stringa di formato `[^=]`.

- *fondi*, che unisce due componenti date di un *merge-find set* dato in una sola.

L'operazione di fusione, in particolare, deve tener conto dell'esigenza di mantenere bassa la complessità computazionale dell'operazione di ricerca, la quale a sua volta dipende dalla profondità degli alberi che costituiscono il *merge-find set*. Questo induce una ben precisa regola di fusione dei due alberi corrispondenti alle componenti da fondere.

4.3.5 Quinta fase

Riconoscimento delle entità e stampa degli enunciati ridondanti

In questa fase si devono definire le seguenti funzioni:

- *CostruisceEntita*, che fonde le definizioni in entità scorrendo gli enunciati di identità e marcando come ridondanti gli enunciati che si dimostrano tali;
- *StampaEnunciatiRidondanti*, che stampa l'elenco degli enunciati di identità ridondanti, nel formato richiesto.

4.3.6 Sesta fase

Stampa delle definizioni complete di ciascuna entità

In questa fase si deve definire la funzione:

- *StampaDescrizioneEntita*, che conta le diverse entità individuate e quindi scorre le definizioni: ogni volta che ne trova una nuova. ne cerca la definizione fondamentale sulla base della lunghezza, la stampa e poi stampa tutte le altre definizioni ad essa equivalenti.

4.3.7 Codice

Capitolo 5

Postoffice

5.1 Problema

In qualità di consulenti per Poste Italiane, dovete scrivere un programma per analizzare le prestazioni attese da un nuovo ufficio postale. Si tratta di misurare quanto a lungo i clienti attenderanno prima di essere serviti. A tale scopo, prima di ristrutturare l'ufficio postale sono stati registrati giorno dopo giorno per alcune settimane gli istanti di ingresso dei clienti nell'ufficio e la durata del servizio da essi richiesto. Questi dati devono venire usati per valutare l'efficacia della nuova struttura, che prevede una coda unica per tutti gli sportelli, sportelli che forniscono tutti i servizi, orari di apertura e chiusura differenziati per i vari sportelli.

Si facciano le seguenti assunzioni sul funzionamento del sistema:

1. gli utenti si mettono in coda appena entrati nell'ufficio;
2. uno sportello aperto serve un utente per volta;
3. appena uno sportello si libera, l'utente in cima alla coda vi accede e viene servito (politica *First-In First-Out*, *FIFO*);
4. se più sportelli sono aperti e liberi, l'utente accede a quello di indice minimo;
5. mentre l'impiegato serve l'utente, lo sportello risulta occupato;
6. appena il servizio è terminato, l'utente esce dall'ufficio postale e lo sportello torna libero;
7. se uno sportello raggiunge l'orario di chiusura mentre sta servendo un utente, attende il termine del servizio per chiudere;

8. la coda provoca un ritardo fra l'istante in cui l'utente entra nell'ufficio e quello in cui comincia ad essere servito; questo ritardo misura l'attesa dell'utente.

5.1.1 Ingresso

L'ingresso si compone di un file che descrive arrivi ed esigenze di servizio degli utenti e un file che descrive numero e orari di servizio degli sportelli.

Il primo file si apre con una riga che riporta il numero degli utenti. Seguono, un utente per riga, l'istante di ingresso dell'utente e la durata in secondi del servizio richiesto, secondo il formato:

hh`:`*mm*`:`*ss* *durata*

dove *hh* rappresenta l'ora di ingresso (un numero di due cifre compreso fra `00` e `23`), *mm* il minuto di ingresso (un numero di due cifre compreso fra `00` e `59`), *ss* il secondo di ingresso (un numero di due cifre compreso fra `00` e `59`) e *durata* rappresenta la durata del servizio in secondi. Per esempio

```
09:05:45 180
```

indica che un utente entra alle nove, cinque minuti e quarantacinque secondi e richiede un servizio della durata di 180 secondi. Si intende che i tempi di ingresso siano in ordine crescente. Utenti diversi possono entrare nello stesso istante, ma l'ordine nella coda rispetta l'ordine con cui compaiono nel file.

Il secondo file si apre con una riga che riporta il numero di sportelli, seguito, uno sportello per riga, dall'orario di apertura, secondo il formato:

hh`:`*mm*`:`*ss*`-`*hh*`:`*mm*`:`*ss*

dove le ore di apertura e chiusura, nel formato già descritto, sono separate da un trattino. Per esempio

```
08:45:00-10:30:00
```

indica che lo sportello apre alle otto e quarantacinque e chiude alle dieci e mezza.

Un esempio di ingresso è riportato nelle Figure 5.1 e 5.2.

5.1.2 Uscita

L'uscita deve riportare in ordine cronologico gli eventi significativi del funzionamento dell'ufficio, cioè:

```
6
08:32:33 120
08:32:58 120
08:33:44 120
08:33:44 120
08:36:00 120
08:40:00 120
```

Figura 5.1: *Esempio di ingresso: file degli utenti*

```
1
08:30:00-12:00:00
```

Figura 5.2: *Esempio di ingresso: file degli sportelli*

- l' ingresso di un utente, nel formato

  hh:*mm*:*ss* `Ingresso utente` *utente*

  dove *hh*, *mm* e *ss* rappresentano l'ora come già descritto e *utente* è l'identificativo dell'utente;

- l'accesso di un utente a uno sportello, nel formato

  hh:*mm*:*ss* `Accesso utente` *utente* `sportello` *sportello*

  dove *sportello* è l'identificativo dello sportello;

- l'apertura o la chiusura di uno sportello, nel formato[1]

  hh:*mm*:*ss* `Apertura sportello` *sportello*

  hh:*mm*:*ss* `Chiusura sportello` *sportello*

- l'uscita di un utente dall'ufficio, nel formato

  hh:*mm*:*ss* `Uscita utente` *utente*

Nell'ultima riga, l'uscita riporta il valor medio dell'attesa degli utenti, espressa in secondi e con due cifre significative.

Un esempio di uscita è riportato nella Figura 5.3.

[1]Si noti che lo sportello apre necessariamente all'ora indicata nel file d'ingresso corrispondente, ma può chiudere in ritardo.

```
08:30:00 Apertura sportello 1
08:32:33 Ingresso utente 1
08:32:33 Accesso utente 1 sportello 1
08:32:58 Ingresso utente 2
08:33:44 Ingresso utente 3
08:33:44 Ingresso utente 4
08:34:33 Uscita utente 1
08:34:33 Accesso utente 2 sportello 1
08:36:00 Ingresso utente 5
08:36:33 Uscita utente 2
08:36:33 Accesso utente 3 sportello 1
08:38:33 Uscita utente 3
08:38:33 Accesso utente 4 sportello 1
08:40:00 Ingresso utente 6
08:40:33 Uscita utente 4
08:40:33 Accesso utente 5 sportello 1
08:42:33 Uscita utente 5
08:42:33 Accesso utente 6 sportello 1
08:44:33 Uscita utente 6
12:00:00 Chiusura sportello 1
Attesa media: 163.17
```

Figura 5.3: *Esempio di uscita: eventi e attesa media*

5.2 Modello

Questo problema comporta in effetti la costruzione di un *simulatore a eventi discreti* dell'ufficio postale. Diremo *entità* ogni elemento che fa parte del sistema (essenzialmente, utenti e sportelli) ed *evento* qualsiasi modifica nello stato di una o più entità. Un evento è quindi associato all'entità, al tipo di modifica del suo stato e all'istante di tempo nel quale avviene.

L'idea di fondo della simulazione a eventi è che alcuni eventi (in questo caso, gli arrivi degli utenti e l'apertura e chiusura degli sportelli) sono generati dall'esterno, mentre altri sono *innescati* dai primi dopo un ritardo più o meno lungo: l'arrivo di un utente produce immediatamente il suo ingresso in coda, l'inizio del servizio di un utente produce la sua uscita dall'ufficio dopo il tempo indicato nel file di ingresso, e così via. Per garantire il rispetto delle relazioni di causa-effetto, gli eventi vengono gestiti in rigoroso ordine cronologico. Quindi, sarà utile conservarli in una struttura che consenta l'estrazione dell'evento con istante minimo e l'aggiunta di nuovi eventi. Tale struttura è la *coda con priorità*.

5.3 Codice

5.3.1 Prima fase

Struttura generale del programma, definizione delle strutture dati, allocazione e deallocazione delle strutture dati ausiliarie

Il codice `postoffice1.c` fornisce le consuete costanti simboliche per definire il tipo `boolean`, gestire gli errori e indicare la lunghezza massima di una riga in lettura (*RIGA*, di valore 255), più

- quattro costanti specifiche per definire il tipo `StatoSportello`, che indica lo stato di uno sportello: `CHIUSO`, `LIBERO`, `OCCUPATO` e `IN_CHIUSURA`;
- una costante per definire l'identificativo di un'entità inesistente: `NO_ENTITY`.

Tramite `typedef`, sono poi definiti i tipi ausiliari:

- *ora*, che è un intero ed esprime un'istante come numero di secondi scorsi dalla mezzanotte;
- *Sportello*, che descrive le caratteristiche di uno sportello (identificativo, ore di apertura e chiusura, stato corrente);
- *InsiemeSportelli*, che contiene l'insieme degli sportelli dell'ufficio postale;
- *Utente*, che descrive le caratteristiche di un utente (identificativo, ora di arrivo, durata del servizio e attesa);
- *InsiemeUtenti*, che contiene l'insieme degli utenti;
- *TipoEvento*, che descrive i vari tipi di eventi, con le relative costanti simboliche: `INGRESSO`, `ACCESSO`, `TERMINE`, `USCITA`, `APERTURA`, `CHIUSURA`;
- *Evento*, che descrive le caratteristiche di un evento (il tipo, l'istante in cui avviene e l'identificatore delle entità, utente e sportello, cui si riferisce); qualora solo una delle due entità sia definita, l'altra assume il valore `NO_ENTITY`.

Viene fornita una libreria di interfaccia per gestire la coda con priorità degli eventi. Nomi e prototipi delle funzioni di tale interfaccia derivano dal Capitolo 7 del testo di Algoritmi e Strutture Dati di A. Bertossi.

```
prioricoda creaprioricoda (heapsize maxsize);
void distruggeprioricoda (prioricoda *pC);
boolean prioricodavuota (prioricoda C);
elemento elemento_min (prioricoda C);
void inserisci (elemento E, prioricoda *pC);
void cancellamin (prioricoda *pC);
int leggiprioricoda (heappos i, prioricoda C);
boolean chiaveminore (heappos e1, heappos e2, prioricoda C);
heappos heapfigliosinistro (heappos i, prioricoda C);
heappos heapfigliodestro (heappos i, prioricoda C);
heappos heappadre (heappos i, prioricoda C);
void restauraheap (heappos e, prioricoda *pC);
```

In particolare, sono forniti i tipi ausiliari:

- *elemento*, che descrive un elemento all'interno della coda, ed è un evento;
- *heap*, che è un vettore di elementi, organizzato secondo la proprietà heap;
- *dimheap*, che rappresenta la dimensione dell'heap ed è un intero;
- *heappos*, che rappresenta una posizione entro un heap ed è un intero;
- *prioricoda*, che descrive la coda con priorità ed è costituita da un heap, dalla sua dimensione corrente e dalla dimensione allocata.

Infine, occorre rappresentare la coda degli utenti, che è una coda *FIFO*. Per farlo, si utilizza la libreria *lista.c*, che funziona come descritto nel Capitolo 2.1.

Il *main* contiene la consueta struttura generale del programma. In particolare, l'allocazione della coda degli eventi viene effettuata prevedendo un massimo numero di eventi simultaneamente presenti nella coda pari al numero totale di eventi. Questo è uguale al triplo del numero di utenti (dato che ognuno entra, accede allo sportello ed esce) più il doppio del numero di sportelli (dato che ognuno apre e chiude).

5.3.2 Seconda fase

sectionheadInterpretazione della linea di comando e caricamento dei dati

In questa fase si devono definire le seguenti procedure:

- *LeggeIstruzioni*, che acquisisce dalla linea di comando i nomi dei file che contengono i dati sugli utenti e sugli sportelli;

- *CaricaUtenti*, che carica le informazioni sugli utenti, gestendo situazioni di errore su apertura file, formati dei dati errato, memoria insufficiente per l'insieme degli utenti;
- *CaricaSportelli*, che carica le informazioni sugli sportelli, gestendo situazioni di errore su apertura file, formati dei dati errato, memoria insufficiente per l'insieme degli sportelli;

5.3.3 Terza fase

Struttura generale della procedura di simulazione, con la stampa degli eventi significativi

In questa fase si deve cominciare a definire la funzione

- *Simula*, che simula il funzionamento dell'ufficio postale; in questa fase, ci si limiti a:
  1. inserire nella coda con priorità degli eventi gli eventi già noti, cioè gli arrivi degli utenti e le aperture e chiusure degli sportelli;
  2. definire i casi possibili[2]
  3. stampare gli eventi corrispondenti a ciascun caso, senza implementare i relativi cambiamenti di stato e senza innescare gli eventi che ne conseguono.

5.3.4 Quarta fase

Prima fase della creazione della libreria di gestione della coda con priorità

In questa fase si definiscano le seguenti funzioni:

- *creaprioricoda*, che crea la coda con priorità, allocando la memoria necessaria e gestendo gli eventuali errori;
- *distruggeprioricoda*, che dealloca la coda con priorità;
- *elemento_min*, che restituisce l'elemento della coda associato all'istante minimo;
- *prioricodavuota*, che verifica se la coda è vuota o no;
- *leggiprioricoda*, che restituisce la chiave (istante) associata a una posizione della coda.

[2]Suggerimento: `switch`.

5.3.5 Quinta fase

Seconda fase della creazione della libreria di gestione della coda con priorità

In questa fase si devono definire le seguenti procedure:

- *inserisci*, che aggiunge un elemento alla coda con priorità aggiornandola in modo che conservi la proprietà heap;
- *cancellamin*, che cancella l'elemento di chiave minima dallo heap;
- *chiaveminore*, che verifica se il primo argomento ha chiave minore del secondo argomento;
- *heapfigliosinistro*, che data la posizione di un elemento dello heap, restituisce la posizione del suo figlio sinistro;
- *heapfigliodestro*, che data la posizione di un elemento dello heap, restituisce la posizione del suo figlio destro;
- *heappadre*, che data la posizione di un elemento dello heap, restituisce la posizione del suo padre;
- *restauraheap*, che istituisce la proprietà heap sul sottoalbero radicato nella posizione data
- *ConvertiIstante*, che converte il tempo di simulazione (secondi scorsi dalla mezzanotte) in una stringa nel formato *hh:mm:ss*.

5.3.6 Sesta fase

Gestione degli eventi di ingresso di nuovi utenti e di accesso degli utenti agli sportelli e uscita degli utenti

In questa fase si aggiunge alla procedura *Simula* il trattamento dei seguenti casi:

- `INGRESSO`, associato all'ingresso di un utente:
  - se gli sportelli sono tutti occupati, l'utente va aggiunto in coda
  - se c'è uno sportello libero, l'utente accede allo sportello di indice minimo ed il corrispondete evento viene aggiunto alla coda con priorità.

- `ACCESSO`, associato all'accesso di un utente allo sportello:
  - occupazione dello sportello, che cambia stato;
  - uscita dell'utente dal sistema dopo la durata associata, cioè innesco di un evento di uscita con istante di tempo pari a quello di accesso più la durata del servizio;
  - calcolo dell'attesa sofferta dall'utente;
- `USCITA`, associatto all'uscita di un utente:
  - se lo sportello che occupava era IN_CHIUSURA, innesca la generazione di un evento di chiusura;
  - se lo sportello che occupava era OCCUPATO, innesca la sua liberazione, cioè il passaggio allo stato di LIBERO
  - se la coda degli utenti non è vuota, innesca un evento di accesso allo sportello dell'utente in cima alla coda;

5.3.7 Settima fase

Gestione degli eventi di apertura, termine del servizio e chiusura degli sportelli

In questa fase si aggiunge alla procedura *Simula* il trattamento dei seguenti casi:

- APERTURA, associato all'apertura di uno sportello:
  - il passaggio dello sportello dallo stato CHIUSO allo stato LIBERO
  - se la coda non è vuota, l'accesso allo sportello dell'utente in cima alla coda (con la generazione del corrispondente evento)
- `TERMINE`, associato alla terminazione del servizio di uno sportello, cioè al fatto che termini il servizio dell'utente corrente e poi chiuda:
  - se lo sportello è libero, innesca la sua chiusura
  - se lo sportello è occupato, provoca il suo passaggio allo stato IN_CHIUSURA
- `CHIUSURA`, associato alla chiusura dello sportello, che fa passare lo sportello nello stato di chiuso.

Questa fase richiede anche di aggiungere le funzioni:

- *leggiprioricoda2*, che restituisce l'identificativo dell'utente associato a un evento, data la sua posizione nella coda;
- *leggiprioricoda3*, che restituisce l'identificativo dello sportello associato a un evento, data la sua posizione nella coda.

Queste funzioni devono inoltre essere usate nei casi in cui due eventi risultano indistinguibili per la chiave principale (l'istante di tempo): a parità di istante di tempo, la coda deve avere come elemento minimo l'evento che riguarda un utente di indice inferiore e a parità di utente quello che riguarda lo sportello di indice minimo.

5.3.8 Ottava fase

Stampa delle statistiche e calcolo dell'attesa media

In questa fase si procede con la definizione della procedura

- *StampaRisultati*, che calcola il tempo di attesa media, sommando i tempi di attesa di tutti gli utenti e dividendo per il numero di utenti, e lo stampa in uscita.

5.3.9 Codice

Capitolo 6

Street

6.1 Problema

Avete ricevuto l'incarico di realizzare un sistema di risposta automatica a interrogazioni sullo stradario di una città. Avete già concluso la prima fase del lavoro, che è consistita nel realizzare uno stradario informatico, costituito da una semplice base di dati con due tabelle. La prima tabella riporta tutti i tronchi stradali della città, definiti attraverso gli indirizzi del loro punto iniziale e terminale. Per "tronco stradale" si intende un tratto di via senza incroci di alcun genere e con un senso di percorrenza ben determinato: se una strada è a doppio senso di marcia, a ogni tratto compreso fra due incroci corrispondono due tronchi stradali con sensi di percorrenza opposti. Un tronco stradale è definito univocamente dall'indirizzo del suo punto iniziale e del suo punto terminale, i quali si affacciano entrambi su un incrocio. La seconda tabella riporta l'indirizzo di ogni punto estremo e il numero identificativo dell'incrocio su cui si affaccia. Un incrocio corrisponde quindi a più punti estremi: uno per ogni tronco stradale che sbocca nell'incrocio stesso o se ne diparte.

La seconda fase del lavoro consiste nel realizzare il sistema in grado di rispondere a interrogazioni. Tali interrogazioni saranno di due tipi:

1. *raggiungibilità*: dati due incroci i e j, indicare se è possibile raggiungere j da i senza attraversare altri incroci intermedi e rispettando i sensi unici;

2. *adiacenza*: dato un incrocio i, elencare gli incroci direttamente raggiungibili da esso.

6.1.1 Ingresso

L'ingresso si compone di tre file. Il primo contiene l'elenco dei tronchi stradali. Comincia con il loro numero. Riga per riga, segue la descrizione di ciascun tronco, che si compone del nome della via, fra virgolette, seguito dai numeri civici del primo e del secondo estremo. Ad esempio:

```
'Vicolo Corto' 1 7
```

Per semplicità, si assuma che il nome di ogni via comprenda al più 80 caratteri.

Il secondo file contiene l'elenco dei punti estremi. Comincia con il loro numero. Riga per riga, segue la descrizione di ogni punto estremo, che si compone del nome della via, fra virgolette, seguito dal numero civico del punto e dall'identificativo numerico dell'incrocio. Per semplicità, detto n il numero degli incroci, i loro identificativi siano i numeri naturali da 1 a n. Ovviamente, i punti estremi che si affacciano sullo stesso incrocio avranno lo stesso identificativo, mentre l'indirizzo completo (via e numero civico) saranno differenti. Ad esempio:

```
'Vicolo Corto' 1 2

'Vicolo Corto' 4 3

'Vicolo Stretto' 1 3
```

Il terzo file contiene le interrogazioni alla base di dati. Si apre con il numero di interrogazioni. Seguono le interrogazioni, riga per riga. Se si tratta di una richiesta di raggiungibilità, il formato è

`R` i j

dove i e j stanno per gli identificativi numerici degli incroci. Se si tratta di una richiesta di adiacenza, il formato è

`A` i

dove i sta per l'identificativo numerico dell'incrocio.

Le Figure 6.1, 6.2 e 6.3 riportano un esempio di ingresso (rispettivamente il file dei tronchi stradali, dei punti estremi e delle interrogazioni). La Figura 6.4 riporta la mappa corrispondente ai primi due file di ingresso.

```
11
'Via Media'      26   2
'Via Media'       1  29
'Vicolo Corto'    1   4
'Vicolo Stretto'  1  21
'Via Di Qua'      1   8
'Via Di Là'       1  11
'Via Larga'       1  33
'Via Larga'      28   2
'Via Larga'      35  45
'Via Larga'      44  30
'Via Media'      31  47
```

Figura 6.1: *Esempio di ingresso: tronchi stradali*

```
22
'Via Media'      26  1
'Via Media'      29  1
'Via Media'      31  1
'Via Di Là'      11  1
'Vicolo Stretto' 21  1
'Via Media'       1  2
'Via Media'       2  2
'Vicolo Corto'    1  2
'Vicolo Corto'    4  3
'Vicolo Stretto'  1  3
'Via Di Qua'      1  3
'Via Di Qua'      8  4
'Via Larga'       1  4
'Via Larga'       2  4
'Via Larga'      28  5
'Via Larga'      33  5
'Via Larga'      30  5
'Via Larga'      35  5
'Via Di Là'       1  5
'Via Media'      47  6
'Via Larga'      44  6
'Via Larga'      45  6
```

Figura 6.2: *Esempio di ingresso: punti estremi*

```
2
R 2 3
A 3
```

Figura 6.3: *Esempio di ingresso: interrogazioni*

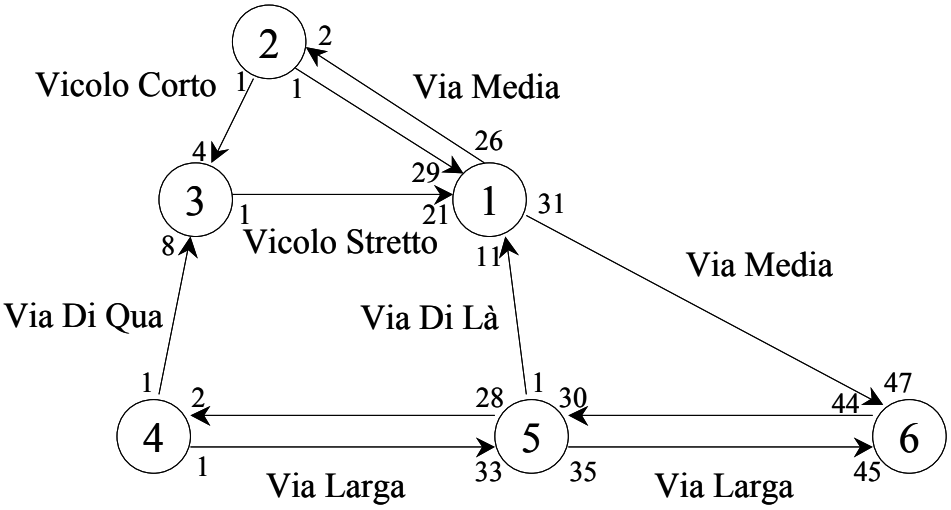

Figura 6.4: *Mappa corrispondente alla rete stradale descritta nelle Figure 6.1 e 6.2*

6.1.2 Uscita

L'uscita deve riportare nell'ordine le interrogazioni, nell'ordine con cui sono state poste, ciascuna seguita dalla relativa risposta. Per le interrogazioni di raggiungibilità, il formato è:

`R` i j

`Sì` (oppure `No`)

Per le interrogazioni di adiacenza, il formato è

`A` i

`Da` i `si raggiungono` j k

Per semplicità, si mantenga questo formato anche se un solo incrocio, o nessun incrocio, è raggiungibile.

La Figura 6.5 riporta l'uscita corrispondente all'ingresso delle Figure 6.1, 6.2 e 6.3.

```
R 2 3
Si'
A 3
Da 3 si raggiungono 1
A 5
Da 5 si raggiungono 1 4 6
R 5 2
No
```

Figura 6.5: *Esempio di uscita*

6.2 Modello

Il problema richiede di gestire relazioni di adiacenza e raggiungibilità fra punti, il modello più ragionevole è rappresentare la rete stradale con un grafo, i cui nodi rappresentano gli incroci, mentre gli archi rappresentano i tronchi stradali. Il grafo è orientato poiché i sensi unici rendono asimmetrica la relazione di adiacenza. Occorre saper dire di ogni coppia di punti se l'uno è raggiungibile o no dall'altro: per svolgere questo compito la rappresentazione del grafo più efficiente dal punto di vista temporale è quella che usa la matrice di adiacenza. Occorre anche saper elencare i punti raggiungibili da un punto dato: per svolgere questo compito la rappresentazione più efficiente è quella

che usa le liste di adiacenza. Realizzeremo entrambe le rappresentazioni durante lo svolgimento.

Da notare anche che la relazione di adiacenza riguarda gli incroci, ma i tronchi stradali che la descrivono sono individuati da indirizzi: ad ogni incrocio (nodo) corrispondono diversi indirizzi.

6.3 Codice

6.3.1 Prima fase

Struttura generale del programma e definizione delle strutture dati

Il *main* segue la struttura generale descritta nell'introduzione. Il codice `Street1.c` dichiara le sue sottoprocedure con i parametri necessari per l'esecuzione.

Oltre alle consuete costanti simboliche per gestire gli errori, la direttiva `#define` fornisce una costante *LUNGHEZZA* (pari a 80) per limitare la lunghezza degli indirizzi. Come di consueto, è definito il tipo enumerativo `boolean`, ma anche il tipo specifico dell'esercizio:

- *Query*, che descrive il tipo di un'interrogazione e può assumere i valori simbolici *ADIACENZA* (`A`) o *RAGGIUNGIBILITA* (`R`).

L'istruzione `typedef` introduce i seguenti tipi ausiliari specifici dell'esercizio:

- *Via* è una stringa di al massimo *LUNGHEZZA* caratteri che descrive il nome di una via;
- *Estremo* descrive un punto estremo di un tronco, attraverso un indirizzo (via e civico) e l'indice dell'incrocio corrispondente;
- *VettoreEstremi* rappresenta un insieme di estremi attraverso un vettore allocato dinamicamente;
- *Tronco* rappresenta un tronco stradale, attraverso una via e i due numeri civici estremi;
- *VettoreTronchi* rappresenta un insieme di tronchi stradali attraverso un vettore allocato dinamicamente;
- *Interrogazione* descrive un'interrogazione alla base di dati, individuata dal tipo di interrogazione e dagli incroci coinvolti (uno o due secondo il tipo di interrogazione)

- *VettoreInterrogazioni* rappresenta un insieme di interrogazioni attraverso un vettore allocato dinamicamente;

. Con la direttiva `#define` si introduce la costante simbolica *NO_INCROCIO*, che corrisponde al risultato negativo nella ricerca di un incrocio a partire da un punto estremo.

Infine, sono fornite due librerie di interfaccia per gestire il grafo, rispettivamente rappresentandolo come matrice di adiacenza e come vettore di liste di adiacenza. Nomi e prototipi delle funzioni di tali interfacce derivano dal Capitolo 9 del testo di Algoritmi e Strutture Dati di A. Bertossi, con una variante significativa: mancano le procedure per aggiungere e togliere nodi al grafo (di conseguenza, la procedura per creare il grafo richiede il numero dei nodi). Il motivo è di semplicità: entrambe le rappresentazioni scelte richiedono operazioni piuttosto macchinose per aggiungere e togliere nodi. D'altra parte, il grafo stradale non cambia durante l'elaborazione ed è piuttosto semplice da costruire.

```
grafo *creagrafo ();
void distruggegrafo (grafo *G)};
boolean grafovuoto (grafo *G);
boolean esistearco (nodo orig, nodo dest, grafo *G)
void insarco (nodo orig, nodo dest, grafo *G)
void cancarco (nodo orig, nodo dest, grafo *G)
```

6.3.2 Seconda fase

Interpretazione della linea di comando e caricamento dei dati

La seconda fase richiede di definire le seguenti procedure:

- *LeggeIstruzioni*, che interpreta la linea di comando, contenente i nomi dei tre file di ingresso;
- *CaricaEstremi*, che deve caricare il file contenente le informazioni sui punti estremi e costruire il relativo insieme;
- *CaricaTronchi*, che deve caricare il file contenente le informazioni sui tronchi stradali e costruire il relativo insieme;
- *CaricaInterrogazioni*, che deve caricare il file contenente le informazioni sulle interrogazioni e costruire il relativo insieme.

6.3.3 Terza fase

Realizzazione della libreria per gestire grafi (matrice di adiacenza)

Questa fase richiede di definire le funzioni necessarie alla creazione e distruzione del grafo:

- *creagrafo*: alloca la matrice di adiacenza[1];
- *distruggegrafo*: rimuove il grafo deallocando la matrice di adiacenza;
- *grafovuoto*: restituisce *TRUE* se il grafo è vuoto, *FALSE* altrimenti;
- *esiste_arco*: risponde *TRUE* se il grafo contiene un arco con gli estremi dati, *FALSE* altrimenti;
- *ins_arco*: aggiunge al grafo un nuovo arco, dati i suoi nodi estremi;
- *canc_arco*: elimina dal grafo un arco individuato dai suoi nodi estremi.

6.3.4 Quarta fase

Conversione dello stradario in grafo

Questa fase definisce la procedura:

- *ConverteStradario.* che cerca gli incroci corrispondenti ai due punti estremi di ciascun tronco e aggiunge al grafo un arco fra i nodi corrispondenti. Per farlo si serve della funzione (che va realizzata):
  - *CercaIncrocio*: determina l'incrocio che corrisponde a un dato punto estremo di un tronco stradale; restituisce *NO_INCROCIO* se nessun incrocio corrisponde a tale punto.

6.3.5 Quinta fase

Procedura di risposta alle interrogazioni

Questa fase definisce la procedura

- *RispondeInterrogazioni*, che risponde alle interrogazioni in base al tipo di interrogazione stessa[2] sfruttando le procedure che restituiscono le informazioni associate al grafo.

[1] Suggerimento: `calloc`.
[2] Suggerimento: `switch`.

6.3.6 Sesta fase

Sostituzione della rappresentazione con matrice di adiacenza con quella con liste di incidenza

Nella settima fase si sostituiscono i file `grafo1.c` e `grafo1.h` con file `grafo2.c` e `grafo2.h` che

- modificano la struttura *grafo* sostituendo alla matrice di adiacenza un insieme (precisamente, un vettore) di liste di incidenza, una per nodo;
- modificano le procedure di gestione del grafo
  - *creagrafo*
  - *distruggegrafo*
  - *esiste_arco*
  - *ins_arco*
  - *canc_arco*

Non sfuggirà all'occhio esperto che la rappresentazione con liste di incidenza permetterebbe di stampare gli incroci adiacenti a quello dato in modo più efficiente che non impiegando la procedura *esiste_arco*, se si potesse scorrere direttamente la lista di incidenza del nodo origine. A rigore, questo non sarebbe consentito in una realizzazione davvero modulare del codice, dato che richiederebbe di accedere alla struttura *G-¿ListaIncidenza* da una funzione che non fa parte della libreria di interfaccia. Per guadagnare in efficienza, occorre o violare l'incapsulamento oppure (secondo la specifica del testo di A. Bertossi) aggiungere una funzione che, dato un nodo, restituisca l'insieme degli archi uscenti da tale nodo, rappresentata opportunamente come lista di archi.

6.3.7 Codice

Capitolo 7

Lucidi

7.1 Problema

Questo pomeriggio si tiene un'importante conferenza. Sfortunatamente, l'organizzatore è una persona molto disordinata, e tiene i lucidi delle presentazioni in un grosso mucchio. Prima della conferenza, quindi, bisogna riordinare i lucidi. Si deve cercare di farlo con poco sforzo, sfruttando il fatto che i lucidi sono numerati. Tuttavia, la posizione del numero di pagina sul lucido può variare e i lucidi sono trasparenti: quindi, i numeri sono tutti visibili, ma non è chiaro su quale lucido sia scritto ciascun numero. D'altra parte, si può dedurlo.

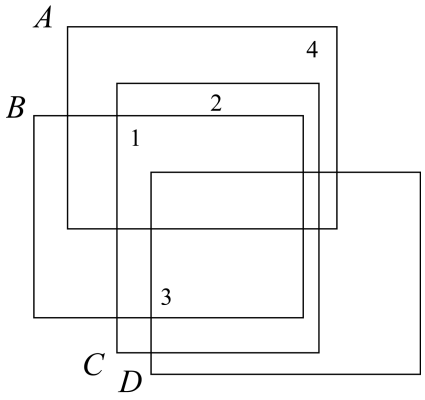

Figura 7.1: *Esempio di disposizione dei lucidi da riordinare*

Nella Figura 7.1, i lucidi sono indicati con i caratteri A, B, C, ecc... e i numeri di pagina con i numeri naturali 1, 2, 3, ecc... Quindi, il lucido D corrisponde sicuramente alla pagina 3, perchè è l'unico che racchiuda il numero

3. Ne deriva che il lucido B corrisponde alla pagina 1, il lucido C alla pagina 2 e il lucido A alla pagina 4: ogni nuovo accoppiamento individuato permette di determinarne altri. Si deve scrivere un programma che automatizzi il procedimento.

7.1.1 Ingresso

L'ingresso fornisce la descrizione del mucchio di lucidi attraverso due file. Il primo comincia con il numero dei lucidi, seguito, riga per riga, dalla posizione di ciascun lucido, nel formato

xmin xmax ymin ymax

dove $(xmin, ymin)$, $(xmin, ymax)$, $(xmax, ymin)$ e $(xmax, ymax)$ sono le coordinate (intere) dei quattro angoli di ciascun lucido. I lucidi sono cioè rettangolari e hanno i lati paralleli agli assi. Ciascuno ha come etichetta una lettera dell'alfabeto, nell'ordine con cui compaiono nell'ingresso. La Figura 7.2 mostra un esempio del primo file di ingresso.

```
4
 6 22 10 20
 4 18  6 16
 8 20  2 18
10 24  4  8
```

Figura 7.2: *Esempio di ingresso: posizioni dei lucidi*

Il secondo file comincia con il numero di pagine, seguito, una pagina per riga in ordine di pagina crescente, dalla posizione del corrispondente numeroi di pagina, nel formato

x y

dove (x, y) sono le coordinate (intere) del numero stesso. Il numero di pagina può cadere in qualsiasi punto della pagina corrispondente, ma mai sul bordo. La Figura 7.3 mostra un esempio del secondo file di ingresso.

7.1.2 Uscita

L'uscita riporta, riga per riga, l'etichetta e il numero di pagina di ciascuno dei lucidi per i quali si è potuto determinare univocamente il numero di pagina, secondo il formato:

x y

```
4
 9 15
19 17
11  7
21 11
```

Figura 7.3: *Esempio di ingresso: posizioni dei numeri di pagina*

Segue il numero di lucidi per i quali non è stato possibile determinare il numero di pagina, nel formato:

n `lucidi non identificati`

La Figura 7.4 mostra un esempio dell'uscita corrispondente ai file di ingresso riportati nelle Figure 7.2 e 7.3.

```
4
 6 22 10 20
 4 18  6 16
 8 20  2 18
10 24  4  8
```

Figura 7.4: *Uscita corrispondente all'ingresso delle Figure 7.2 e 7.3*

7.2 Modello

Il problema richiede di scoprire una corrispondenza biunivoca fra lucidi e numeri di pagina, data una relazione di compatibilità fra loro: ogni lucido è compatibile con le pagine il cui numero compare all'interno del perimetro del lucido. Qualora un numero sia contenuto nel perimetro di un solo lucido, esso gli deve corrispondere; viceversa, qualora un lucido racchiuda nel proprio perimetro un solo numero, esso gli deve corrispondere. Essendo la corrispondenza biunivoca, ogni accoppiamento riduce le compatibilità fra gli altri lucidi e numeri di pagina. Al termine, o tutte i lucidi e i numeri sono stati accoppiati, oppure rimane un nocciolo duro di lucidi e numeri compatibili fra loro.

Il modello più naturale per questo problema consiste in un grafo bipartito in cui i vertici di una sponda rappresentano i lucidi e quelli dell'altra i numeri di pagina, mentre i lati rappresentano la compatibilità fra lucido e pagina (cioè si ha un lato ogni volta che le coordinate del numero cadono nel rettangolo individuato dalle coordinate del lucido). Costruito il grafo

che rappresenta la relazione di compatibilità, ogni vertice con un solo lato incidente rappresenta un elemento la cui corrispondenza è individuata, e per il quale si può quindi fissare un accoppiamento. Questa operazione comporta di eliminare ogni altra relazione di compatibilità (cioè ogni lato) per i due elementi accoppiati, e quindi eventualmente di far emergere nuovi accoppiamenti obbligati.

7.3 Stesura del codice

7.3.1 Prima fase

Struttura generale del programma, definizione delle strutture dati

Il *main* segue la struttura generale descritta nell'introduzione. Poiché il codice richiederà diverse librerie ausiliarie, corrispondenti a file di dichiarazione e di definizione, e poiché alcune dichiarazioni fondamentali sono comuni a tutti i file, viene fornito un file `defs.h` che contiene le consuete costanti simboliche per gestire gli errori e il consueto tipo enumerativo `boolean`. Il codice `Lucidi1.c` fornisce attraverso la direttiva `#define` una costante *LUNGHEZZA* (pari a 80) per limitare la lunghezza dei nomi dei file. L'istruzione `typedef` introduce i seguenti tipi ausiliari specifici dell'esercizio:

- *Lucido*, che descrive un lucido attraverso il nome e le quattro coordinate dei suoi angoli (come dati del problema) e il numero di pagina corrispondente (da determinare come risultato);
- *VettoreLucidi* rappresenta un insieme di lucidi attraverso un vettore allocato dinamicamente;
- *Pagina*, che descrive una pagina attraverso il numero e le due coordinate dei suoi estremi;
- *VettorePagine*, che rappresenta un insieme di pagine attraverso un vettore allocato dinamicamente.

Inoltre è fornita una libreria ausiliaria per la gestione di grafi, rappresentati attraverso le liste di incidenza. La scelta di questa rappresentazione è motivata dal fatto che la proprietà fondamentale su cui si basa l'algoritmo è l'adiacenza fra vertici corrispondenti a lucidi e vertici corrispondenti a pagine. Tale libreria coincide con quella utilizzata nell'esercizio precedente, con l'aggiunta di un'informazione utile: il *grado* di ciascun vertice, rappresentato con un vettore di numeri interi e accessibile attraverso un'opportuna funzione di libreria. Una seconda differenza rilevante è che il grafo da rappresentare è

non orientato. Questo significa che contiene lati (privi di orientamento) anziché archi (orientati) e che un lato deve comparire nelle liste di incidenza di entrambi i suoi estremi. Quindi ogni lato corrisponde a due strutture dati diverse, che continueremo a indicare come "archi" e che avranno orientamento opposto, cioè origine e destinazione scambiati. L'aggiunta e la cancellazione di lati dovranno tenere conto di questa duplicazione.

```
grafo *creagrafo ();
void distruggegrafo (grafo *G)};
boolean grafovuoto (grafo *G);
int grado (vertice v, grafo *G);
boolean esistelato (vertice v1, vertice v2, grafo *G)
void inslato (vertice v1, vertice v2, grafo *G)
void canclato (vertice v1, vertice v2, grafo *G)
```

7.3.2 Seconda fase

Interpretazione della linea di comando, caricamento dei dati, restituzione dei risultati

La seconda fase deve realizzare le seguenti procedure:

- *LeggeIstruzioni*, che interpreta la linea di comando, ricevendo il nome dei due file di ingresso che contengono lucidi e numeri di pagina;
- *CaricaLucidi*, che carica l'elenco dei lucidi da individuare;
- *CaricaPagine*, che carica le posizioni dei numeri di pagina;
- *StampaAccoppiamenti*, che fornisce in uscita la soluzione del problema indicando le coppie lucido-pagina riconosciute e gli eventuali lucidi non identificati.

7.3.3 Terza fase

Realizzazione della libreria di gestione del grafo non orientato

La terza fase dell'esercizio prevede di realizzare le procedure di interfaccia per la gestione del grafo, servendosi della libreria `listaarchi.c` e `listaarchi.h` che gestisce liste di archi (da usare per rappresentare gli insiemi di archi incidenti in ogni vertice). Precisamente, occorre realizzare le procedure:

- *creagrafo*, che alloca la memoria necessaria a rappresentare un grafo con un numero dato di vertici e nessun lato

- *distruggegrafo.* che libera la memoria usata per descrivere un grafo.
- *grafovuoto*: restituisce *TRUE* se il grafo è vuoto, *FALSE* altrimenti;
- *grado*: fornisce il grado (numero di lati incidenti) per il vertice dato;
- *esiste_lato*: risponde *TRUE* se il grafo contiene un lato con gli estremi dati, *FALSE* altrimenti;
- *ins_arco*: aggiunge al grafo un nuovo lato, dati i suoi vertici estremi;
- *canc_arco*: elimina dal grafo un lato individuato dai suoi vertici estremi.

7.3.4 Quarta fase

Costruzione del grafo di compatibilità fra lucidi e numeri di pagina

In questa fase si deve realizzare la procedura

- *CostruisceGrafoCompatibilità*, che aggiunge un lato al grafo per ciascuna coppia lucido-pagina compatibile (numero di pagina che cade entro il lucido).

7.3.5 Quinta fase

Determinazione degli accoppiamenti univoci

Nell'ultima fase bisogna realizzare la procedura

- *AccoppiaLucidiPagine*, che accoppia lucidi e pagine per i quali si possa dimostrare la corrispondenza.

A tale scopo, scorre i vertici del grafo, determinando quelli con un solo lato incidente: tali vertici vanno accoppiati stabilmente con l'unico vertice a loro adiacente. Bisogna però anche procedere a eliminare ogni altro lato incidente in entrami i vertici e ripetere il procedimento finché ha esito. Per farlo è opportuno conservare in una lista i vertici per i quali si deve realizzare l'accoppiamento. A tale scopo è fornita la libreria di interfaccia `listavertici.c` e `listavertici.h`. La lista va gestita come coda (cioè First-In-First-Out, ovvero *FIFO*).

7.3.6 Codice

Capitolo 8

Alea

8.1 Problema

La società di gestione di un casinò vuole introdurre nuove tecnologie di sorveglianza nei suoi giochi, per contenere il più possibile gli imbrogli. In particolare, per i giochi che richiedono il lancio di dadi, il risultato non verrà più riconosciuto solamente dal croupier, ma anche da un sistema di visione robotica, che dovrà confermare la lettura umana e registrarla. Questo comporta l'uso di una telecamera, posta sopra il tavolo da gioco, e di un programma che analizzi l'immagine del tavolo verde determinando il numero che appare su ciascun dado, qualunque sia il numero e la posizione dei dadi.

8.1.1 Ingresso

L'ingresso programma consiste nell'immagine di un lancio di dadi, codificata in un file di testo. La prima riga del file riporta le dimensioni dell'immagine secondo il formato

w h

dove w è l'ampiezza e h l'altezza.

Le successive h righe contengono w caratteri l'una (oltre al carattere di fine riga), corrispondenti ai pixel dell'immagine, secondo il formato:

- `.` per i pixel dello sfondo del tavolo (verde)
- `*` per i pixel dei dadi (bianchi)
- `X` per i pixel dei pallini (neri)

Si noti che l'immagine di un pallino può comprendere uno o più pixel e l'immagine di un dado può avere dimensioni differenti e non essere perfettamente quadrata, a causa di fenomeni di distorsione ottica.

La Figura 8.1 riporta un esempio di ingresso.

```
30 15
..............................
..............................
..............*...............
...*****......****............
...*X***.....**X***...........
...*****....***X**............
...***X*.....****.............
...*****.......*..............
..............................
........***........******.....
.......**X****.....*X**X*.....
......*******......******.....
.....****X**.......*X**X*.....
........***........******.....
..............................
```

Figura 8.1: *Esempio di ingresso: codifica dell'immagine ripresa da telecamera*

8.1.2 Uscita

L'uscita deve riportare nella prima riga il numero di dadi, nella seconda i risultati del lancio, cioè il numero che compare su ciascun dado, in ordine decrescente. La Figura 8.2 riporta un esempio di uscita corrispondente all'ingresso della Figura 8.1.

```
4
4 2 2 1
```

Figura 8.2: *Esempio di uscita corrispondente all'ingresso della Figura 8.2*

8.2 Modello

La griglia di pixel del tavolo da gioco è rappresentata in modo naturale da una matrice di caratteri. Tuttavia, il problema richiede strutture ausiliarie per riconoscere i dadi e i pallini sui dadi. Ogni pallino su un dado e ogni

dado (pallini compresi) costituisce una regione connessa nell'immagine. Al contrario, dadi diversi nella stessa immagine e pallini diversi (anche sullo stesso dado) disegnano regioni disgiunte tra loro. Per convenzione, consideriamo adiacenti due pixel che si toccano in verticale o in orizzontale, cioè appartenenti alla stessa riga e a colonne vicine oppure alla stessa colonna e a righe vicine, mentre non consideriamo adiacenti due pixel che condividono un angolo.

Per rappresentare una relazione di adiacenza diretta e quindi di connessione anche indiretta, risulta naturale impiegare come struttura dati un grafo non orientato: i vertici del grafo corrispondono ai pixel, i lati alle coppie di pixel adiacenti.

Sarà conveniente procedere attraverso due fasi: nella prima fase verranno identificati i dadi, nella seconda i pallini appartenenti a ciascun dado. Nella prima fase, i vertici corrispondono ai pixel appartenenti ai dadi (pallini compresi) e i lati alle coppie di pixel adiacenti: ogni componente connessa corrisponde a un dado. Nella seconda fase, i vertici corrispondono ai soli pixel appartenenti ai pallini e i lati alle coppia di pixel adiacenti: ogni componente connessa corrisponde a un pallino. Per conoscere il numero di pallini su ogni dado basta sapere a quale dado appartenga ogni pixel, informazione ricavata nella prima fase.

8.3 Codice

8.3.1 Prima fase

Struttura generale del programma e definizione delle strutture dati

Il *main* segue la struttura generale descritta nell'introduzione. Viene fornito un file `defs.h` che contiene le consuete costanti simboliche per gestire gli errori e il consueto tipo enumerativo `boolean`. Il codice `Alea1.c` fornisce attraverso la direttiva `#define` le costanti

- *LUNGHEZZA* (pari a 80) per limitare la lunghezza dei nomi dei file;
- *NO_COMP* (pari a 0) per esprimere la componente connessa di cui fa parte un vertice non ancora assegnato ad alcuna componente.

L'istruzione `typedef` introduce i seguenti tipi ausiliari specifici dell'esercizio:

- *tiponodo*, che definisce il tipo di un pixel (*TAVOLO*, *DADO* o *PALLINO*);

- *Immagine*, che definisce un'immagine come matrice dinamica bidimensionale di *tiponodo*;

Infine, è fornita una libreria per la gestione di grafi orientati (il grafo dato non è orientato, ma possiamo servirci della libreria a patto di sostituire ad ogni lato una coppia di archi opposti).

8.3.2 Seconda fase

Interpretazione della linea di comando, caricamento e deallocazione delle strutture dati principali

In questa fase, devono essere definite le le seguenti funzioni:

- *LeggeIstruzioni*, che interpreta la linea di comando acquisendo il nome del file di ingresso;
- *CaricaImmagini*, che alloca e inizializza la matrice che conserva l'immagine da elaborare;
- *DistruggeImmagine*, che dealloca la struttura dati per conservare l'immagine.

8.3.3 Terza fase

Realizzazione della libreria di gestione del grafo orientato

L'obiettivo di questa fase è quello di completare le procedure per la creazione, la distruzione e la stampa del grafo. Si deve quindi procedere realizzando i corpi delle tre seguenti funzioni:

- *creagrafo*, che alloca la memoria necessaria per un grafo vuoto;
- *distruggegrafo*, che dealloca tutte le strutture che descrivono un grafo;
- *ins_nodo*
- *ins_arco* .

8.3.4 Quarta fase

Costruzione del grafo che rappresenta i dadi

La quarta fase prevede la definizione funzione

- *CostruisceGrafoDadi*, che scorre l'immagine e aggiunge al grafo un nodo per ogni pixel appartenente ad un dado o a un pallino e una coppia di archi opposti per ogni coppia di nodi corrispondenti a pixel adiacenti.

8.3.5 Quinta fase

Determinazione dei dadi

In questa fase si devono definire le seguenti funzioni:

- *DFS* (Depth-First Search): la funzione deve effettuare la visita in profondità del grafo ed operare come segue: si parte da un vertice s non visitato e lo si visita quindi si sceglie un vertice non scoperto adiacente ad s. Da s si attraversa quindi un percorso di vertici adiacenti (visitandoli) finchè non si incontra un vertice già visitato. Appena si resta "bloccati", quando cioè tutti gli archi da un vertice sono stati visitati, si torna indietro (backtracking) di un passo (vertice) nel percorso attraversato e si aggiorna il vertice s al vertice corrente dopo il passo all'indietro. Infine si ripete il processo ripartendo da un vertice non ancora scoperto.

- *CompConnesse*: questa funzione deve scorrere la lista dei nodi del grafo per individuare, utilizzando la funzione DFS, le componenti connesse all'interno del grafo stesso. utilizzando la funzione DFS.

- *StampaComponenti*: il compito di questa procedura è quello di stampare tutte le componenti appartenenti all'insieme individuato.

8.3.6 Sesta fase

Identificazione dei pallini appartenenti a ciascun dado

La sesta fase prevede l'aggiunta del corpo della funzione

- *IdentificaPallini*, che chiamerà come sottoprocedure, da dichiarare senza ancora definirle, la costruzione del grafo dei pallini e la stampa del loro numero

8.3.7 Settima fase

Costruzione dei vari grafi che rappresentano i pixel appartenenti ai pallini di ciascun dado e stampa del loro numero per ciascun dado

La fase conclusiva prevede il completamento del programma con l'aggiunta del corpo della funzione

- *CostruisceGrafoPallini*, che aggiunge un nodo al grafo GP per ogni nodo del grafo GD che stia nel dado d e sia un pixel-pallino. Inoltre la funzione deve anche aggiungere un arco allo stesso grafo per ogni coppia di nodi corrispondenti a pixel adiacenti.
- *StampaNumeroPallini*.

8.3.8 Codice