5 Dicembre 2016

Programmazione M-Z
Ingegneria e Scienze Informatiche - Cesena
A.A. 2016-2017

Elaborato 9

Data di sottomissione: entro la mezzanotte del 18 Dicembre 2016.
Formato di sottomissione: un file compresso con nome `Elaborato9.zip`, contenente un unico file sorgente con nome `snake.c`

Specifiche:

- Sviluppare funzioni di libreria per poter gestire l'**oggetto serpente** nel gioco *snake*.
- Viene fornita l'implementazione dell'intero gioco, tranne l'implementazione della libreria `snake.c`. L'implementazione fa uso della libreria `curses`.
- I prototipi delle funzioni da implementare sono dichiarati nell'header `snake.h` e allegati alle specifiche.
- Il serpente è rappresentato tramite una **lista doppiamente concatenata**.
- Le funzioni di libreria in `snake.c` si occupano di:
  - creare una lista concatenata che rappresenti un serpente di lunghezza 1,
  - distruggere una lista concatenata che rappresenti un serprente,
  - spostare il serpente in una direzione (up, down, right, left),
  - aggiungere una nuova testa al serpente in una direzione specificata (up, down, right, left) rispetto alla testa attuale,
  - rimuovere un determinato numero di parti del serpente, a partire dalla coda,
  - invertire il serpente (i.e. la coda diventa la testa e viceversa),
  - ritornare la posizione attuale della testa attuale del serpente,
  - ritornare la posizione attuale di una parte del corpo del serpente,

  - verificare che il serpente non si sia *annodato* (i.e. che la lista non contenga ripetizioni delle stesse coordinate).

- Le funzioni di libreria prendono in input un puntatore `s` al primo nodo della lista doppiamente concatenata che rappresenta il serpente.
  - Il mondo in cui si sposta il serpente è uno spazio toroidale rappresentato da una matrice di dimensione `rows`×`cols`.
  - Le posizioni attuali del corpo del serpente sono salvate nei nodi della lista concatenata.
  - Il serpente cresce di lunghezza aggiungendo nodi alla lista.
- Per semplificare le implementazioni, è possibile evitare di gestire gli eventuali casi di allocazione non riuscita di memoria: le dimensioni del gioco rendono alquanto improbabile che si verifichi questa possibilità.
- Differenze rispetto alla versione 2.0 (in sintesi)
  - Essendo il mondo toroidale, il serprente attraversa i bordi per sbucare dalla parte opposta. Non è quindi più possibile morire a causa di collisioni con il bordo dello spazio di gioco.
  - Mangiare cibo cattivo causa il dimezzamento della dimensione del serpente ma tale cibo non ha effetti su un serpente di lunghezza 1 (baby serpente). L'unico modo per evitare la terminazione del gioco è evitare che il serpente si annodi (e.g. evitare che la testa tocchi qualche parte del corpo).
  - La velocità del serpente è direttamente legata alla sua lunghezza (entro certi limiti): più è lungo più si sposta velocemente.
  - E' stato introdotto un nuovo tipo di cibo, *creepy food*, che causa un cambiamento della direzione di movimento del serpente (si allontana con disgusto): in questi casi il serpente viene *ribaltato* (la coda diventa la testa) e prosegue per una diversa direzione.
  - E' possibile mettere in pausa il gioco e farlo ripartire non appena è terminato senza dover rilanciare l'eseguibile.

Vincoli:

- Le implementazioni devono aderire perfettamente ai prototipi e alle specifiche fornite.
- Le eventuali funzioni di utility della libreria e variabili globali devono essere *nascoste* all'esterno.

Suggerimenti:

- Un aspetto da considerare prima di passare all'implementazione è dove posizionare la testa del serpente nella lista concatenata: in testa o coda?
- La funzione di di spostamento nella direzione `dir` produce un nuovo set di coordinate per l'oggetto serpente. Come per le implementazioni precedenti, il nuovo set di coordinate è equivalente al set di coordinate ottenuto con la seguente procedura:

  1. aggiungiamo una nuova testa nella direzione `dir` rispetto alla vecchia testa,
  2. rimuoviamo la coda.

- Per verificare se il serpente è annodato è sufficiente verificare che le coordinate della testa non siano ripetute nella lista.
- Lo spostamento avviene adesso in un mondo toroidale. E' quindi necessario sapere quali sono le dimensioni del mondo, *rows* e *cols*, per poter correttamente spostare il serpente attraverso i bordi. Il corpo del serpente può unicamente occupare le posizioni (i, j), dove $i \in [0, rows - 1]$ e $j \in [0, cols - 1]$.

  - Spostamento in verticale. Abbiamo un problema se la posizione i è minore o maggiore-uguale di *rows*. E' sufficiente applicare la seguente formula per riportarla nel range corretto:

    $$i' = (i + rows) \mod rows$$

  - Spostamento in orizzontale. Abbiamo un problema se la posizione j è minore o maggiore-uguale di *cols*. E' sufficiente applicare la seguente formula per riportarla nel range corretto

    $$j' = (j + cols) \mod cols$$

- La funzione `snake_reverse()` inverte completamente il serpente. Ad esempio, una lista con posizioni

  {(1,1), (1,2), (2,2), (2,3)}

  dopo una chiamata alla `snake_reverse()` diventa

  {(2,3), (2,2), (1,2), (1,1)}

- Le funzioni di libreria `snake_move()`, `snake_reverse()`, `snake_increase()` e `snake_decrease()` possono eventualmente causare la modifica del puntatore al primo elemento della lista doppiamente concatenata che rappresenta il serpente. Tale modifica deve essere direttamente gestita all'interno di tali funzioni. Per questo motivo, i prototipi delle suddette funzioni sono dichiarati in modo da prendere come argomento **l'indirizzo di memoria del puntatore alla testa della lista** (e non, quindi, l'indirizzo di memoria della testa della lista). E' necessario prestare particolare attenzione a tali funzioni. Si suggerisce di implementare per prima la funzione `snake_reverse()` e testarne il funzionamento al di fuori del gioco.

```c
#ifndef SNAKE_H
#define SNAKE_H

enum direction {UP, DOWN, LEFT, RIGHT};

struct position {
 unsigned int i;
 unsigned int j;
};

struct snake {
 unsigned int i;
 unsigned int j;
 struct snake *next;
 struct snake *prev;
};
/*
 * Creates a snake's head in a world of size rows*cols;
 */
struct snake *snake_create(unsigned int rows, unsigned int cols);
/*
 * Destroys the snake data structure.
 */
void snake_kill(struct snake *s);
/*
 * Returns the (position of the) snake's head.
 */
struct position snake_head(struct snake *s);
/*
 * Returns the (position of the) i-th part of the snake body.
 *
 * Position 0 is equivalento the snake's head. If the snake
 * is shorter than i, the position returned is not defined.
 */
struct position snake_body(struct snake *s, unsigned int i);
/*
 * Returns 1 if the snake crosses himself, 0 otherwise.
 */
int snake_knotted(struct snake *s);
/*
 * Moves the snake one step forward in the dir direction
 * into a toroidal world of size rows*cols.
 */
void snake_move(struct snake **s, enum direction dir);
/*
 * Reverses the snake.
 *
 * The tail of the snake is now the new head.
 */
void snake_reverse(struct snake **s);
/*
 * Increases the snake length.
 *
 * This is equivalent to:
 * - add a new head in the dir direction wrt the old head.
 */
void snake_increase(struct snake **s, enum direction dir);
/*
 * Removes from the tail of the snake decrease_len parts.
 */
void snake_decrease(struct snake **s, unsigned int decrease_len);

#endif
```