Informatica Generale
Homework 2/2017: Dinamicamente in Successione

Ivano Salvo – Sapienza Università di Roma – 8/5/2017

Esercizio 1 Scrivere una funzione C di prototipo:

```
int** spirale(int n)
   /* PREC: n>0 */
```

che preso come parametro di input un numero intero $n > 0$ alloca una matrice dinamica di dimensione $n \times n$, caricandola a *spirale*.

Si mette in alto a sinistra (in posizione [0][0]) il numero 1 e poi, andando in senso *orario* verso il centro della matrice, si dispongono in ordine gli altri numeri 2,3,4, ... fino a n^2.

Esempi: Ecco ad esempio le matrici a spirale di lato 1, 2, 3 e 4:

```
1        1  2        1  2  3         1   2   3   4
         4  3        8  9  4        12  13  14   5
                     7  6  5        11  16  15   6
                                    10   9   8   7
```

Esercizio 2 Sia P una lista non vuota che contiene ordinatamente i primi k numeri primi ($k > 0$). Scrivere il codice di una funzione C di prototipo

```
int nthPrime(listFirstLast P, int n);
   /* PREC: P non vuota, contiene ordinatamente
      i primi k numeri primi */
```

che restituisce l'n-esimo numero primo. Se $n \leq k$, `nthPrime` si limita a ritornare il contenuto dell'n-esimo nodo di `P`. Altrimenti, oltre a ritornare l'n-esimo numero primo, modifica P aggiungendovi in coda altri $n - k$ numeri primi.

Il tipo `listFirstLast` è semplicemente il tipo `lista` visto a lezione, in cui invece del puntatore alla testa, avete un puntatore a un record contenente un puntatore alla testa ed uno alla coda[1].

Suggerimento: Potrebbe essere utile, nell'ottica di strutturare il codice, definirsi una funzione `void nextP(listFirstLast P)`, che aggiunge in coda a `P` il prossimo numero primo nella sequenza dei primi.

Esercizio 3 Consideriamo la seguente successione di numeri naturali: $u_0 = 1, u_1 = 2$ e $u_k(k > 1)$ è il *minimo* numero naturale maggiore di u_{k-1} che si può scrivere in *modo unico* come somma di due distinti precedenti numeri della successione. La successione sopra definita comincia con $1, 2, 3, 4, 6, 8, 11, 13, 16, 18, 26, 28, \ldots$.

Osservate che u_4 non può essere 5, in quanto $5 = 2+3 = u_1+u_2$, ma anche $5 = 1 + 4 = u_0 + u_3$. Viceversa solo $u_1 + u_3 = 2 + 4$ danno come somma 6.

Scrivere una funzione C di prototipo:

```
int nextU(listDCFirstLast U){
     /* PREC: U contiene ordinatamente
        i primi k>=2 elementi della successione u */
```

che, sotto la precondizione che `U` contenga *ordinatamente* i primi $k > 2$ elementi della successione u, calcola il $k + 1$-esimo, lo restituisce come risultato e lo aggiunge in coda a `U`.

Il tipo `listDCFirstLast` è il tipo *lista doppiamente concatenata*, che estende il tipo `listFirstLast` facendo in modo che ogni nodo oltre a un pointer al successivo abbia un pointer al precedente[2].

Suggerimenti Potete ovviamente seguire molte strade. Per facilitare la ricerca del $k + 1$-esimo elemento u_{k+1} (una volta noti $u_0, u_1, \ldots, u_k$) osservate che necessariamente si ha che $u_k + 1 \leq u_{k+1} \leq u_{k-1} + u_k$. Infatti, essendo la successione strettamente crescente, è ovvio che $u_{k-1} + u_k$ si scrive in modo unico come somma di due precedenti (in quanto questa somma è strettamente maggiore di ogni altra somma di coppie di precedenti) e quindi la ricerca del prossimo elemento ha sempre un esito positivo (alla peggio $u_{k-1} + u_k$, il che, tra l'altro, dimostra banalmente che la successione è infinita).

---

[1] vedi file `list.h`. Alcune funzioni di utilità sono in `list.c`

[2] vedi file `listDC.h`. Alcune funzioni di utilità sono in `listDC.c`

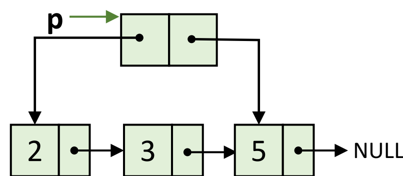
Note Pratiche e Concettuali

L'uso di liste con puntatori alla testa e alla coda (negli Esercizi **2** e **3**) è motivato dal fatto che queste permettono di accedere direttamente all'ultimo nodo e all'ultimo valore memorizzato, il che può rendere più efficienti la ricerca del prossimo elemento da inserire, così come il successivo inserimento.

Analogamente, le liste doppiamente concatenate potrebbero facilitare la scrittura di un algoritmo lineare (in k) per verificare se un numero può essere il $k+1$-esimo elemento da inserire nella successione dell'Esercizio **3**, in quanto esse permettono di percorrere le liste sia in avanti che all'indietro.

Per esemplificare ulteriormente cosa devono fare le funzioni richieste, spero risultino eloquenti le Figure 1 e 2.

a) Sia **`p`** come in figura



b) risultato in memoria ***dopo*** l'esecuzione di **`nthPrime(6,p);`** (che torna **`13`**)

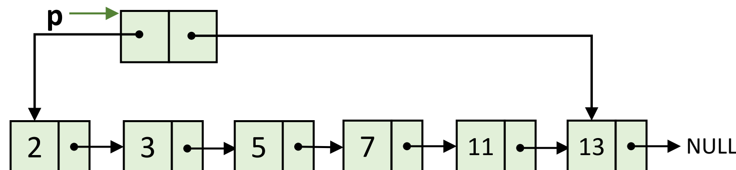


Figura 1: Esempio di esecuzione della funzione `nthPrime` (Esercizio **2**).

Potete usare le funzioni contenute nei files `list.c` e `listDC.c`, che forniscono le operazioni base di inserimento, creazione e stampa di liste. Per compilare il tutto, al solito, dovreste usare i comandi (supponendo le funzioni richieste più eventuali funzioni ausiliarie nei files `Esercizio2.c` ed `Esercizio3.c`):

```
gcc -std=c99 list.c Esercizio2.c hw2-main.2.c
gcc -std=c99 listDC.c Esercizio3.c hw2-main.3.c
```

Risultato in memoria ***dopo*** l'esecuzione di **`u=initializeU()`**(**a**) e
dopo l'esecuzione di **1**, e **4** chiamate di **`nextU(u)`** (× per NULL)

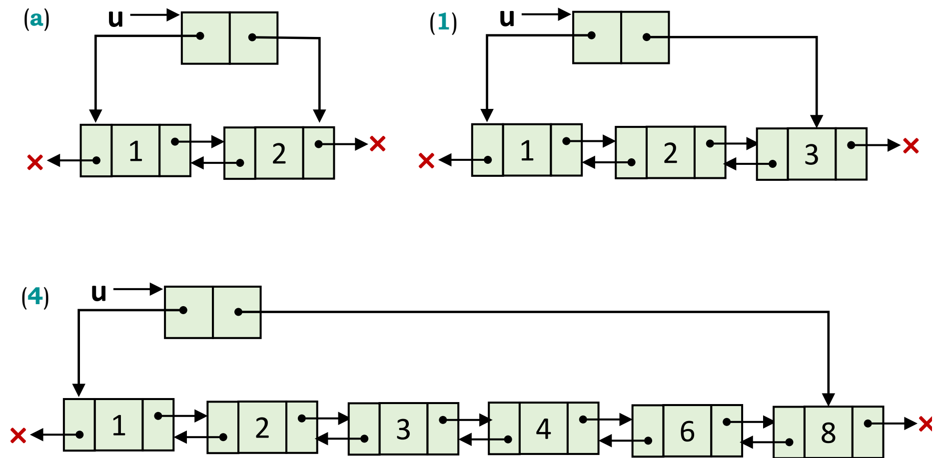


Figura 2: Esempio di esecuzione della funzione `nextU` (Esercizio **3**).

Dovete includere le librerie `list.h`, `listDC.h`, `stdlib.h`, e `stdio.h` ovunque necessario. Questo non causa problemi. Se andate a vedere le definizioni di `list.h` vedrete che le definizioni stanno dentro un costrutto:

```
#ifndef LIST_H
#define LIST_H
typedef
...
#endif
```

questo permette di compilare separatamente i files, ed evitare doppie definizioni. Tutte le librerie standard sono ovviamente fatte in questo modo.

Osservate infine che le "nostre" librerie vanno incluse con i comandi:

```
#include "list.h"
#include "listDC.h"
```

mentre le librerie di sistema con le parentesi angolate. Questo informa il compilatore di cercare nella directory corrente invece che nelle directory in cui sono archiviate le librerie standard.