Esercizio

Scrivere un programma che legge una sequenza di numeri interi e li dispone in una lista concatenata, in modo che la lista sia sempre ordinata in senso crescente. Il programma deve stampare la lista a ogni nuovo inserimento. I numeri devono essere letti dallo *standard input*. La lettura si interrompe quando si legge il valore 0, che non deve essere inserito nella lista.

Svolgimento

Poiché il testo stabilisce quale struttura dare ai dati, l'algoritmo da applicare è immediato: a ogni nuovo valore letto, il programma deve percorrere la lista fino a trovare un numero maggiore di quello da inserire, e deve inserire il nuovo valore prima di quello trovato (casi particolari: se la lista è vuota, il nuovo numero sarà il primo, e unico; se il numero da inserire è maggiore di tutti quelli già presenti, dovrà essere inserito in fondo alla lista).

Il testo stabilisce di usare una lista concatenata come struttura astratta dei dati. Nella rappresentazione concreta, ogni nodo porterà un puntatore all'elemento successivo. L'ultimo avrà un puntatore nullo al successivo (nel seguito, userò a volte il termine "catena" come sinonimo di "lista concatenata").

Il tipo corrispondente si può definire in C come segue:

```
typedef struct nodo Nodo;
struct nodo {
  int valore;
  Nodo * prox;
};
```

Ci servirà una variabile che punti al primo elemento della lista:

```
Nodo * testa;
```

L'operazione più importante che il programma dovrà svolgere sulla catena è l'inserimento di un nuovo elemento; nel caso in esame, l'inserimento può avvenire in qualsiasi posizione della catena.

All'inizio dell'esecuzione, quindi, la catena sarà vuota, e la variabile *testa* avrà valore *NULL*. Dobbiamo assegnare esplicitamente questo valore:

```
testa = NULL;
```

La struttura portante del programma sarà un ciclo, nel quale, a ogni iterazione, si legge il nuovo valore e lo si inserisce al posto opportuno:

```
scanf ("%d", &n);
while ( n != 0 ) {
    testa = inserisci(n, testa);
    stampa(testa);
    scanf ("%d", &n);
}
```

In questo frammento, suppongo di avere una funzione *inserisci*, che riceve come argomenti un puntatore a una lista e un numero, e inserisce il nuovo numero al suo posto; la funzione restituisce il valore del puntatore al primo elemento della nuova lista (nel caso che il nuovo elemento sia più piccolo di tutti quelli già presenti, dovremo aggiungerlo in testa alla catena, e quindi il valore della variabile *testa* cambierà); suppongo anche di avere una funzione *stampa*, che riceve come argomento un puntatore a una lista e stampa il contenuto della lista, in ordine.

NOTA METODOLOGICA: nel corso dello sviluppo di un programma, conviene spesso fingere di avere già scritto, o comunque di avere già a disposizione, funzioni che svolgono compiti specifici, come l'inserimento in questo caso. Ci si può così concentrare sulla struttura generale del programma, senza perdersi subito nei dettagli.

La funzione di stampa è la più semplice: deve percorrere la lista, e stampare via via il campo *valore* del nodo che sta attraversando.

```
void stampa (Nodo * t) {
    while ( t != NULL ) {
        printf("%d ", t->valore);
        t = t->prox;
    }
    return;
}
```

La funzione di inserimento richiede un po' più di attenzione. Innanzitutto, poiché abbiamo deciso di passare il numero da inserire come semplice valore, la funzione deve preoccuparsi di allocare lo spazio in memoria necessario per il nodo della catena che dovrà ospitarlo:

```
nuovo = (Nodo *) malloc (sizeof(Nodo));
if ( nuovo == NULL ) exit(EXIT_FAILURE);
nuovo->valore = n;
nuovo->prox = NULL;
```

Poiché queste quattro istruzioni svolgono un compito ben definito, che potrebbe essere utile anche in altri punti di un programma più complesso di quello che

stiamo analizzando, è buona norma spostarle in una funzione a sé, che verrà invocata dalla funzione di inserimento. La nuova funzione ha quindi il compito di allocare lo spazio necessario per un nuovo elemento di tipo *Nodo*, assegnare al suo campo *valore* un valore *n*, e di produrre come risultato un puntatore a questo nuovo elemento. La funzione avrà quindi questa forma:

```
Nodo * nuovonodo (int n) {
    Nodo * np;
    np = (Nodo *) malloc (sizeof(Nodo));
    if ( np == NULL ) exit(EXIT_FAILURE);
    np->valore = n;
    np->prox = NULL;
    return np;
}
```

La funzione di inserimento dovrà dichiarare una variabile di tipo "puntatore a Nodo", e assegnarle il risultato dell'invocazione della funzione *nuovonodo*.

```
Nodo * nuovo;
nuovo = nuovonodo (n);
```

Ora possiamo cominciare a percorrere la catena per trovare il punto in cui inserirlo. Ci serviamo di un puntatore ausiliario che si sposterà lungo la catena. Poiché ci sono due possibili vie d'uscita dal ciclo (troviamo un elemento maggiore del nuovo numero, oppure arriviamo in fondo alla lista) definiamo una variabile, concettualmente di tipo *booleano*, che permette di distinguere i due casi. Trattiamo a parte il caso speciale in cui la lista è vuota.

```
if (testa == NULL) {
    return nuovo;
}
```

Se c'è almeno un elemento, dobbiamo scorrere gli elementi fino a trovare il posto in cui inserire il nuovo arrivato. Come ci accorgiamo di avere raggiunto il punto di inserimento? Confrontando il nuovo valore con quello dell'elemento corrente della catena; quando questo valore è maggiore o uguale di quello nuovo, dobbiamo inserire. Incontriamo qui un problema determinato dalla natura delle liste concatenate semplici: quando troviamo un valore maggiore di quello da inserire, il puntatore ausiliario ha già superato il punto di inserimento, e nelle catene semplici non si può tornare indietro; il problema si pone anche quando il nuovo valore è maggiore di tutti quelli già presenti; in questo caso, il nuovo valore va inserito in fondo alla catena, ma lo scopriamo quando il puntatore ausiliario vale *NULL*, e non possiamo tornare indietro. Per risolvere questa difficoltà, ci serviremo di un secondo puntatore ausiliario, che seguirà, a un "passo" di distanza, il primo.

```
int inserito;
Nodo * cur;  /* Primo puntatore ausiliario */
Nodo * bcur; /* Secondo puntatore ausiliario */
inserito = 0; /* Interpretiamo 0 come 'falso' */
cur = testa->prox;
bcur = testa;
while ( cur != NULL && inserito == 0 ) {
    if ( nuovo->valore <= cur->valore ) {
```

Stiamo trattando il caso in cui *cur* punta a un elemento maggiore del nuovo numero, che dovrà quindi essere inserito prima di *cur*. Occorre fare attenzione alla manipolazione dei puntatori e all'ordine degli assegnamenti.

```
        nuovo->prox = cur;
        bcur->prox = nuovo;
        inserito = 1;
    }
    else /* Passiamo all'elemento successivo */
        cur = cur->prox;
        bcur = bcur->prox;
}
```

Dobbiamo ancora trattare il caso in cui *cur* percorre tutta la catena senza trovare un elemento maggiore del nuovo. In questo caso, all'uscita dal ciclo, *cur* ha il valore *NULL*, e possiamo trattare a parte la circostanza.

```
if ( cur == NULL ) {
    nuovo->prox = NULL;
    bcur->prox = nuovo;
}
```

Infine, la funzione restituisce il valore del puntatore alla testa della catena.

```
return testa;
```

Non resta che cucire insieme i pezzi, e aggiungere il contorno. Il risultato è il seguente.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct nodo Nodo;

struct nodo {
```

```
  int valore;
  Nodo * prox;
};

void stampa (Nodo * t) {
    while ( t != NULL ) {
        printf("%d ", t->valore);
        t = t->prox;
    }
    printf ("\n");
    return;
}

Nodo * nuovonodo (int n) {
        Nodo * np;
        np = (Nodo *) malloc (sizeof(Nodo));
        if ( np == NULL ) exit(EXIT_FAILURE);
        np->valore = n;
        np->prox = NULL;
        return np;
    }

Nodo * inserisci(int n, Nodo * t) {
    Nodo * nuovo;

    nuovo = nuovonodo (n);

    if (t == NULL) { /* Catena vuota */
        return nuovo;
    }

    if (nuovo->valore <= t->valore) {
        nuovo->prox = t;
        return nuovo;
    }

    Nodo * cur;
    Nodo * bcur;
    int inserito = 0; /* Interpretiamo 0 come 'falso' */
    cur = t->prox;
    bcur = t;
    while ( cur != NULL && inserito == 0 ) {
        if ( nuovo->valore <= cur->valore ) {
            nuovo->prox = cur;
            bcur->prox = nuovo;
            inserito = 1;
```

```
        }
        else { /* Passiamo all'elemento successivo */
            cur = cur->prox;
            bcur = bcur->prox;
        }
    }

    if ( cur == NULL ) {
        printf("cur ha raggiunto il fondo\n");
        nuovo->prox = NULL;
        bcur->prox = nuovo;
    }

    return t;
}

int main (int argc, char * argv[]) {
    int n;
    Nodo * testa;

    testa = NULL;

    scanf ("%d", &n);
    while ( n != 0 ) {
        testa = inserisci(n, testa);
        stampa(testa);
        scanf ("%d", &n);
    }

    exit(EXIT_SUCCESS);
}
```