Laboratorio di Algoritmi e Strutture Dati

Docente: Camillo Fiorentini

11 dicembre 2007

Esercizio 1

L'obiettivo è di scrivere un programma `list.c` per gestire insiemi dinamici di interi (insiemi che variano nel tempo). Il programma deve leggere da standard input una sequenza di istruzioni secondo il formato nella tabella, dove n è un intero. I vari elementi sulla riga sono separati da uno o più spazi. Quando una riga è letta, viene eseguita l'operazione associata; le operazioni di stampa sono effettuate sullo standard output, e ogni operazione deve iniziare su una nuova riga.

Istruzione in input	**Operazione**
`+` n	Se n non appartiena all'insieme lo inserisce, altrimenti non compie alcuna operazione
`-` n	Se n appartiene all'insieme lo elimina, altrimenti non compie alcuna operazione
`?` n	Stampa un messaggio che dichiara se n appartiene all'insieme
`c`	Stampa il numero di elementi dell'insieme
`p`	Stampa gli elementi dell'insieme
`o`	Stampa gli elementi dell'insieme in ordine crescente
`d`	Cancella tutti gli elementi dell'insieme
`f`	Termina l'esecuzione

Si assume che l'input sia inserito correttamente. Conviene scrivere le istruzioni di input in un file `in.txt` ed eseguire il programma redirigendo lo standard input.

Struttura dati

Per rappresentare l'insieme usare una *lista di interi*. L'ordine in cui gli elementi di un insieme compaiono nella lista è irrilevante. Dalle specifiche delle operazioni si deduce che gli elementi della lista sono distinti.

Per rappresentare un elemento della lista definire la struttura

```
struct element {
    int info;  // valore dell'elemento
    struct element *next;   // indirizzo del prossimo elemento
};
```

Conviene porre la definizione

```
typedef  struct element  element;
```

che consente di usare il tipo `element` come sinonimo del tipo `struct element`.

Il campo `next` di un elemento è l'indirizzo del prossimo elemento della lista e vale `NULL` se l'elemento è l'ultimo della lista. Ricordarsi che il valore iniziale di una variabile non è definito, quindi se una variabile di tipo puntatore deve avere valore iniziale `NULL`, l'inizializzazione va fatta *esplicitamente*.

Struttura della funzione `main()`

In `main()` va definita una variabile `head` di tipo `element*` che rappresenta la lista. Quindi:

- Se `head` vale `NULL`, `head` rappresenta la lista vuota.
- Se `head` è diverso da `NULL`, `head` è l'indirizzo al primo elemento (testa) della lista.

```
int main(){
   // DEFINIZIONE VARIABILI LOCALI
   // INIZIALIZZAZIONE DELLA VARIABILE head

   while((c=getchar()) != 'f'){
    /* c e' il prossimo carattere letto da standard input
       Il ciclo termina quando c e' il carattere 'f' */
       switch(c){
            case '+':
              // CODICE PER OPERAZIONE '+ n'
              break;
            case '-':
              // CODICE PER OPERAZIONE '- n'
              break;
            ...
           // ALTRI CASI
           ...
           } // end switch
    } // end while
   // LA LISTA PUO' ESSERE CANCELLATA
   return 0;
} // end main()
```

La funzione `main()` va completata gradualmente, man mano che si definiscono le funzioni descritte sotto.

Funzioni da definire

1. Scrivere il codice della funzione

```
element* insert(int n, element* h)
```

che inserisce in testa alla lista `h` un nuovo elemento contenente `n` e restituisce la lista ottenuta (ossia, l'indirizzo del primo elemento della nuova lista).

Conviene inserire il nuovo elemento in *testa* in modo tale che il numero di operazioni richieste sia costante (non dipende dalla lunghezza della lista). Non è necessario trattare il caso della lista `h` vuota a parte.

2. Scrivere il codice della funzione

```
void printList(element* h)
```

che stampa tutti gli elementi della lista `h`.

Per provare le funzioni scritto finora. scrivere in `main()` il codice dell'operazione '`p`' e dell'operazione '`+n`', tralasciando per il momento il controllo di appartenenza di `n` alla lista `h`.

3. Scrivere il codice della funzione

```
element* find(int n, element* h)
```

che cerca l'intero `n` nella lista `h`. Se `n` è nella lista la funzione restituisce l'indirizzo dell'elemento corrispondente, altrimenti restituisce `NULL`.

A questo punto è possibile scrivere in `main()` il codice dell'operazione '`?n`' e completare il codice di '`+n`' (l'inserimento viene fatto solo se `n` non è già nella lista).

4. Scrivere il codice della funzione

```
element* delete(element* x, element* h)
```

che cancella l'elemento di indirizzo `x` nella lista `h` e restituisce la lista ottenuta. Si *assume* che `x` sia l'indirizzo di un elemento della lista (quindi `h` non può essere la lista vuota).

Distinguere i casi in cui l'elemento da cancellare `x` è il primo della lista e il caso in cui `x` non è il primo. In quest'ultimo caso, occorre attraversare la lista per determinare l'indirizzo dell'elemento che precede `x`.

Aggiungere in `main()` il codice per l'operazione '`-n`'.

5. Per contare gli elementi, anziché scrivere una funzione che determina la lunghezza della lista conviene definire in `main()` una variabile contatore `count` il cui valore è il numero di elementi nella lista.

Modificare il codice scritto in modo che il valore di `count` sia sempre aggiornato e aggiungere in `main()` l'operazione '`c`'.

6. Scrivere il codice della funzione

```
void destroy(element* h)
```

che cancella tutti gli elementi della lista `h`.

Aggiungere in `main()` l'operazione '`d`'.

7. Per completare il programma `list.c`, rimane da fare la stampa degli elementi della lista in ordine crescente. Dato che non esistono algoritmi efficienti per ordinare liste, occorre copiare gli elementi della lista in un array, ordinare l'array usando un algoritmo efficiente quindi stampare l'array ordinato usando una opportuna funzione `printArray()`.

Più in dettaglio, occorre per prima cosa definire una funzione

```
int* listToArray(element* h, int n)
```

che, data una lista `h` contenente `n` interi, restituisce l'indirizzo di un array di interi creato dinamicamente contenente gli elementi della lista.

Quindi si ordina l'array creato da `listToArray()` e si stampa l'array. A questo punto l'array non serve più e la memoria da esso occupata può essere rilasciata.

Per ordinale l'array di interi, si può usare una delle funzioni in `sort.c`. Occorre includere in `list.c` il file `sort.h` (**non** va incluso `sort.c`) che contiene i prototipi delle funzioni principali di `sort.c`

```
#include  "sort.h"    // header file contenente prototipi
```

I programmi `list.c` e `sort.c` vanno compilati insieme:

```
gcc list.c sort.c
```

Si noti che in `list.c` e in `sort.c` non possono esserci funzioni con lo stesso nome (in particolare, in `sort.c` non può essere definita una funzione `main()`).

Per evitare di ricompilare tutte le volte il programma `sort.c` (che non richiede modifiche), conviene creare il file oggetto `sort.o` con il comando

```
gcc -c  sort.c
```

quindi creare l'eseguibile facendo

```
gcc list.c sort.o
```

oppure

```
gcc -Wall list.c sort.o
```

Esercizio 2

Consideriamo un cristallo che si evolve nel tempo nel modo seguente. All'istante $t = 0$ il cristallo è costituito da un solo elemento quadrato di lato unitario. All'istante $t + 1$, il cristallo ha al centro un elemento quadrato di lato unitario e, a ciascuno dei quattro vertici del quadrato è adiacente un cristallo ottenuto al tempo t.

Ad esempio, indicando con '`*`' un elemento quadrato di lato unitario del cristallo e con con '`.`' un quadrato unitario vuoto, i cristalli ottenuti ai tempi $t = 0, 1, 2$ sono:

```
                                *.*.*.*
                                .*...*.
            *.*                 *.*.*.*
  *         .*.                 ...*...
            *.*                 *.*.*.*
                                .*...*.
                                *.*.*.*

 t=0        t=1                   t=2
```

L'obiettivo è quello di costruire una matrice di `char` che rappresenta il cristallo ottenuto all'istante t. Ogni cella della matrice rappresenta un quadrato unitario dello spazio che può essere pieno o vuoto, come nell'esempio precedente. Per rappresentare lo stato di una cella si possono usare le definizioni

```
#define PIENA '*'
#define VUOTA '.'
```

a. Scrivere una funzione ricorsiva

```
int latoCristallo(int t)
```

dove $t \geq 0$, che calcola la misura del lato l del cristallo al tempo t.
Ad esempio, per $t = 0$ il valore di l è 1; per $t = 1$ l vale 3.

b. Scrivere una funzione

```
char **creaMatrice(int n)
```

che crea dinamicamente una matrice quadrata di `char` di lato n (ossia, una matrice $n \times n$) in cui tutti gli elementi risultano vuoti.

c. Scrivere una funzione

```
void printMatrice(char **M, int n)
```

che stampa il contenuto della matrice quadrata $n \times n$ passata come primo parametro.

d. Il cristallo va costruito ricorsivamente. Non essendo possibile in C passare una sottomatrice di una matrice, occorre passare alla funzione l'*intera* matrice, specificando le righe e le colonne della sottomatrice da considerare.

La funzione ricorsiva

```
void  crist(char **M, int r0, int c0, int l)
```

costruisce il cristallo di lato l nella sottomatrice della matrice M composta dagli elementi $M[r][c]$ tali che:

$$r_0 \leq r < r_0 + l \qquad c_0 \leq c < c_0 + l$$

La funzione deve riempire la sottomatrice in esame in base alla definizione di cristallo. Assumere che gli indici r_0, c_0,l siano corretti (ossia, $M[r][c]$ è effettivamente un elemento della matrice M).

Suggerimento: Il caso base si ha quando $l = 1$ (cristallo al tempo 0). Nel passo induttivo, occorre riempire l'elemento centrale della sottomatrice e fare quattro chiamate ricorsive per riempire le quattro sottomatrici che descrivono i cristalli adiacenti ai vertici dell'elemento centrale (cristalli costruiti al tempo immediatamente precedente a quello attuale).

e. Scrivere una funzione per fare la chiamata principale della funzione del punto precedente. La funzione deve avere intestazione

```
void cristallo(char **M, int l)
```

dove l è la misura del lato di un cristallo e M è una matrice quadrata $l \times l$. La funzione deve riempire la matrice in modo che rappresenti il cristallo di lato l.

Se l'esercizio è stato svolto correttamente, le seguenti linee di codice stampano il cristallo al tempo t, dove il valore di t è letto da standard input.

```
char **matrix;
int t, lato;
scanf("%d", &t); //  legge il tempo
```

```
if(t>= 0){
  lato = latoCristallo(t);    // dimensione della matrice
  matrix = creaMatrice(lato); // crea matrice per rappresentare il cristallo
  cristallo(matrix, lato);   // costruisce il cristallo avente lato assegnato
  printMatrice(matrix, lato);    // stampa la matrice
}
```

Con $t = 3$ deve essere stampato:

```
*.*.*.*.*.*.*.*
.*...*...*...*.
*.*.*.*.*.*.*.*
...*.......*...
*.*.*.*.*.*.*.*
.*...*...*...*.
*.*.*.*.*.*.*.*
.......*.......
*.*.*.*.*.*.*.*
.*...*...*...*.
*.*.*.*.*.*.*.*
...*.......*...
*.*.*.*.*.*.*.*
.*...*...*...*.
*.*.*.*.*.*.*.*
```

Se non si usa la ricorsione, la soluzione non è banale.

Completare l'esercizio inserendo istruzioni per liberare lo spazio occupato dalla matrice.

Nota

Essendo i cristalli di lato l uguali, nel Punto d si può in realtà fare una sola chiamata ricorsiva per determinare il primo cristallo C_1 di lato $l - 1$; per gli altri tre cristalli è sufficiente copiare C_1 al posto giusto.