Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

mercoledì 8 novembre 2017

Lista di interi

L'obiettivo è di scrivere un programma `list.c` per gestire insiemi dinamici (che variano nel tempo) di interi . 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 nell'ordine inverso
`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 scritte 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

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

che cerca l'intero `n` nella lista `h`. Se `n` è nella lista la funzione restituisce 1 altrimenti restituise 0.

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(int n, element* h)
```

che cancella l'elemento con valore `n` nella lista `h` e restituisce la lista ottenuta. Distinguere i casi in cui l'elemento da cancellare è il primo della lista e il caso in cui non è il primo. Se `n` non è nella lista, la funzione non deve fare nulla.

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 inverso.

Per accedere ad un elemento della lista bisogna scorrerla a partire dalla testa e, dato un puntatore ad un elemento, non è possibile ottenere un puntatore all'elemento precedente se non scorrendo nuovamente la lista dalla testa.

Per evitare questo problema, si può quindi procedere in due modi.

- Si possono copiare gli elementi della lista in un array, quindi stamparlo al contrario. 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 stampa l'array al contrario. A questo punto l'array non serve più e la memoria da esso occupata può essere rilasciata.

- In alternativa si può usare la ricorsione. Più precisamente, occorre definire una funzione

```
void printInv( element* h )
```

che scorre la lista richiamando sè stessa ad ogni nuovo elemento e si conclude stampando l'elemento stesso.

Variante 1: uso di puntatori a puntatori

Le funzioni definite ai punti 1 e 4 modificano, in genere, la lista passata per argomento, restituendo un puntatore alla testa della nuova lista. Quando vengono chiamate, è pertanto necessario ri-assegnare la testa della lista con il valore restituito dalle funzioni:

```
element *head;
...
head = insert( 5, head );
...
head = delete( 3, head );
```

Usando i puntatori a puntatori è possibile modificare direttamente la testa della lista. I nuovi prototipi saranno:

```
void insert( int n, element **h );
void delete( int n, element **h );
```

e le chiamate andranno modificate come segue:

```
insert( 5, &head );
void delete( 3, &head );
```

Variante 2: lista come struttura

Invece di dichiarare una variabile globale `count` è possibile rappresentare una lista attraverso una variabile strutturata, formata da un puntatore alla testa e da un intero che indica il numero di elementi della lista:

```
typedef struct {
    int count;              /* numero di elementi */
    element *head;          /* indirizzo del primo elemento */
} list;
```

Invece di dichiarare `element *head` bisognerà quindi dichiarare una variabile `list l`.

Naturalmente, vanno modificate tutte le funzioni precedenti: anzichè passare come argomento `element *h`, dovrete passare `list *l`; inoltre per accedere alla testa della lista o al numero dei suoi elementi, dovrete scrivere `l -> head` e `l -> count`, rispettivamente.