ESERCIZIO n° 26

Scrivere un modulo C che implementi l'astrazione stack.

Soluzione

Per operare in un caso concreto, supporremo di dover realizzare uno stack di numeri, di tipo *double*: l'adattamento ad altri tipi è, peraltro, immediato. Per implementare lo stack, utilizziamo un vettore, che chiameremo `val`, con un indice (referenziante la prima posizione libera) di nome `sp` (Stack Pointer). Naturalmente, con queste scelte la nostra realizzazione è soltanto un'approssimazione del concetto di stack, in quanto, mentre teoricamente non vi sono limiti al numero di elementi inseribili, nel nostro caso non potranno essere introdotti più di `MAXDIM` elementi, corrispondentemente alla dimensione massima del vettore usato come supporto.

Per questo, oltre alle classiche funzioni di inserimento ed estrazione di un elemento, decidiamo di fornire anche due funzione di utilità, denominate `isFull()` e `isEmpty()`, che restituiscano un valore (intero) di tipo vero/falso a seconda che lo stack sia, rispettivamente, pieno o vuoto.

Al fine di garantire la consistenza della struttura dati, provvediamo a dichiarare `static` le due variabili globali, in modo da renderle invisibili fuori dal file corrente: l'accesso alla struttura dovrà così obbligatoriamente avvenire attraverso le due funzioni predisposte, `push()` e `pop()`, mentre lo stato della struttura sarà accessibile tramite le due funzioni di utilità sopra citate.

A proposito di `pop()`, occorre stabilire cosa accade nel caso che venga invocata quando lo stack è vuoto: mentre nel caso della `push()`, infatti, può bastare un messaggio d'errore, perché tale funzione è di tipo void e non restituisce nulla, nel caso della `pop()` occorre decidere se restituire qualcosa e, se sì, cosa. Per semplicità, in questa sede si è scelto di restituire un valore qualunque, -333.333: in una realizzazione reale, viceversa, sarebbe opportuno predisporre una soluzione più sofisticata, quale, ad esempio, l'introduzione di un'ulteriore variabile globale con funzione di flag, da settare quando qualcosa va storto, e leggibile tramite un'apposita funzione `stack_result()`. Questa modifica è lasciata per esercizio.

```
     /*   File STACK.C:   implementa l'astrazione stack        */

#include <stdio.h>
#define MAXDIM 100

static int      sp=0;           /* variabili esterne, permanenti e   */
static double   val[MAXDIM];    /* invisibili, che costituiscono le  */
                                /* strutture dati                    */

void push(double f)
{
     if (sp<MAXDIM)
          val[sp++]=f;
     else
          printf("\nErrore: stack pieno; %g non inseribile\n",f);
}

double pop(void)
{
     if (sp>0)
          return val[--sp];
```

```
     else {
          printf("\nErrore: stack vuoto\n");
          return -333.333;
          }
}

int isFull(void)
{
     return (sp==MAXDIM);
}

int isEmpty(void)
{
     return (sp==0);
}
```

Osserviamo che i test all'interno di `push()` e `pop()` corrispondono esattamente a chiedersi se lo stack sia, rispettivamente, pieno (nel qual caso è impossibile la push) o vuoto (nel qual caso è impossibile la pop): poiché abbiamo predisposto due funzioni che realizzano esattamente tali test, il codice di push e pop potrebbe anche essere riscritto, forse in nodo più leggibile, come segue.

```
          /*       variante di push() e pop()       */

void push(double f)
{
     if (!isFull())
          val[sp++]=f;
     else
          printf("\nErrore: stack pieno; %g non inseribile\n",f);
}

double pop(void)
{
     if (!isEmpty())
          return val[--sp];
     else {
          printf("\nErrore: stack vuoto\n");
          return -333.333;
          }
}
```

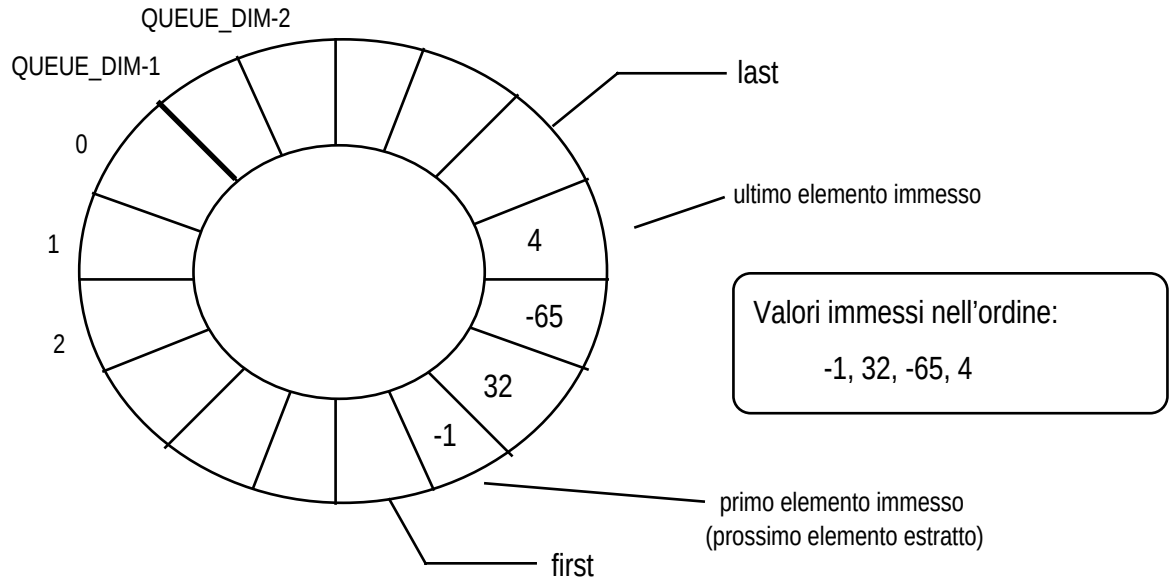
Per provare questo modulo, approntiamo un piccolo programma di prova, che consente di immettere valori (introducendo 0.0 per terminare) e successivamente li estrae, visualizzandoli. Il file "stack.h" riunisce le *dichiarazioni* delle quattro funzioni esportate (cioè rese visibili all'esterno) da "stack.c".

```
                /*   UN PROGRAMMA DI PROVA */

#include <stdio.h>
#include "stack.h"      /* include le dichiarazioni delle funzioni */

void main(void)
```

```
{
 double f;

 do {
  printf("\nValore:\t");
  scanf("%lf",&f);
  push(f);
 } while(f!=0.0);

 while (!isEmpty())
  printf("\nValore:\t%f", pop() );

 printf("\nLa struttura dati è vuota.\n");
}
```

ESERCIZIO n° 27
Scrivere un modulo C che implementi l'astrazione coda circolare.

Soluzione
Come nel caso precedente, assumiamo di operare con numeri, di tipo *double*. Per implementare la coda, utilizziamo ancora un vettore, che chiameremo `val`, gestito però, stavolta, con *due indici*, di nome `last` e `first`, referenzianti rispettivamente la fine dalla coda (più esattamente, la prima posizione libera in fondo alla coda, dove verrà inserito il prossimo elemento) e l'inizio della stessa (più precisamente, la prima posizione davanti all'elemento da estrarre). La figura sottostante illustra la situazione su un esempio concreto, unitamente alla numerazione delle celle del vettore (da 0 a `QUEUE_DIM`-1)



La realizzazione proposta è limitata a un massimo di `QUEUE_LENGTH` elementi, corrispondenti alla dimensione massima del vettore usato come supporto (`QUEUE_DIM`) *diminuita di uno*: questo fatto è una diretta conseguenza della scelta di denotare con `last` e `first` le due celle limitrofe alle coda vera e propria. Se da un lato, infatti, ciò implica che almeno una cella resterà sempre vuota, dall'altro consente di scrivere le funzioni di inserimento ed estrazione in modo più semplice, ed è stato perciò preferito in questa sede. Inoltre, forniremo le usuali funzioni di utilità `isFull()` e `isEmpty()`, con il solito significato. Sempre per garantire la consistenza della struttura dati, dichiariamo `static` le variabili globali, in modo da renderle invisibili fuori dal file corrente e assicurare che l'accesso alla struttura avvenga soltanto attraverso le funzioni predisposte, che per ragioni di uniformità continuiamo a chiamare `push()` e `pop()`, sebbene non si tratti più di uno stack.

Dal momento che la coda va gestita circolarmente, tutti gli incrementi e i decrementi degli indici devono avvenire *modulo* `QUEUE_DIM`: ciò implica controllare, a ogni operazione, se si è "a cavallo" della fine del vettore (denotata in figura da una riga più marcata), provvedendo in tal caso a correggere i calcoli appunto di una quantità `QUEUE_DIM`.

Per questa ragione, le espressioni di verifica se la coda è piena (in `isFull`) o vuota (in `isEmpty`) contengono delle espressioni condizionali, che effettuano il calcolo della differenza fra gli indici o in modo "normale" o con la correzione suddetta a seconda della situazione attuale. In ogni caso, al di là dei tecnicismi usati nelle formule, concettualmente `isFull` è vera se la differenza fra gli indici (eventualmente corretta come sopra) raggiunge `QUEUE_DIM`, mentre `isEmpty` è vera se tale differenza si riduce a 1, come si può constatare da una lettura attenta del codice.
In conseguenza dell'organizzazione adottata, l'inserzione di un nuovo elemento (push) richiede che *prima* si riempia la cella corrente (indicata da `last`), e *poi* si incrementi l'indice, mentre, dualmente, l'estrazione (pop) richiede che *prima* si incrementi l'indice `first` e *poi* si estragga l'elemento da esso indicato. Ciò si riflette immediatamente nel tipo di incremento adottato, che è un *post-incremento* (`last++`) nel primo caso, e un *pre-incremento* (`++first`) nel secondo. Per comprendere bene il codice di `pop`, tenere presente che la condizione dell'`if` viene valutata (e quindi `first` incrementato) sia che poi la stessa risulti vera, sia che risulti falsa.

```
     /* File QUEUE.C:   implementa l'astrazione coda circolare  */

#include <stdio.h>
#define QUEUE_DIM 100
#define QUEUE_LENGTH QUEUE_DIM-1

static int last=0;              /* variabili globali che gestiscono  */
static int first=QUEUE_LENGTH;  /* la struttura dati (invisibili)    */
static double val[QUEUE_DIM];

int isFull(void)
{
   return (last>first ? last-first : QUEUE_DIM+last-first)>=QUEUE_DIM;
}

int isEmpty(void)
{
   return (last>first ? last-first : QUEUE_DIM+last-first) <= 1;
}

void push(double el)
{
     if (!isFull())
       {
          val[last++]=el;
          if(last>=QUEUE_DIM) last-=QUEUE_DIM;
       }
     else
          printf("\nErrore: coda piena. %lg non inseribile\n",el);
}

double pop(void)
{
     if (!isEmpty())
       {
          if (++first>=QUEUE_DIM) first-=QUEUE_DIM;
          return val[first];
```

```
      }
    else
      {
         printf("\nErrore: coda vuota.\n");
         return -555.555;
      }
}
```

Avendo mantenuto identici i nomi delle funzioni, come programma di prova si può riusare, *senza nessuna modifica*, quello dell'esercizio precedente: solo, in luogo dello header "stack.h" andrà posto il nuovo header "queue.h", che, peraltro, risulta identico all'altro.
Per usare l'una o l'altra struttura dati basterà quindi sostituire (in Turbo C) il file "stack.c" con "queue.c" nella descrizione del progetto, e rifare il *make* del tutto. Lo studente attento potrà notare che il programma in sé *non* sarà ricompilato, ma soltanto collegato (dal linker) alle nuove funzioni definite in "queue.c", questo, sì, compilato sul momento.

Questo esercizio, unitamente al precedente, mostra in tutta evidenza l'estrema utilità di *modularizzare i problemi*, *incapsulando* i dettagli delle soluzioni e delle realizzazioni dei singoli sottoproblemi, in modo da ridurre al minimo i punti di contatto fra essi, ottenendo in cambio grande *flessibilità* e *facilità di adottare nuovi comportamenti* al mutare delle situazioni.
Come risultato, immettendo gli stessi elementi rispettivamente nello stack o nella coda, con lo stesso programma principale, essi saranno restituiti e visualizzati in un diverso ordine, corrispondente alla diversa *politica di accesso* realizzata dai due tipi di dato astratto (LIFO nel caso dello stack, FIFO nel caso della coda).