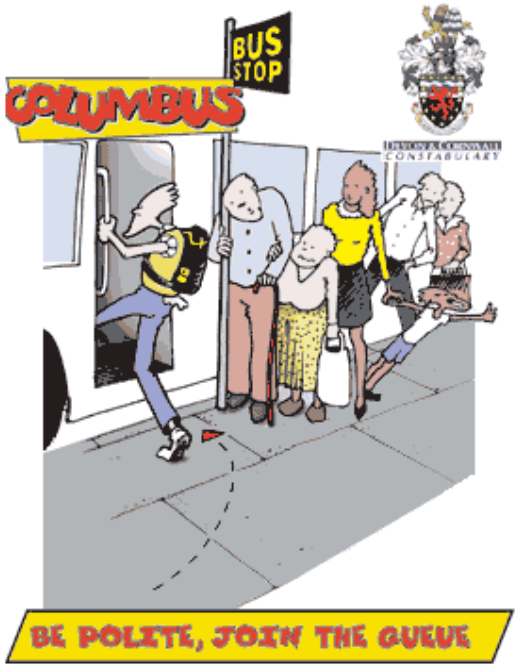
In una coda



gli inserimenti si fanno alla fine e le cancellazioni all'inizio!
First In First Out

CODA: i requisiti

Una coda (queue) è un ADT in cui le operazioni sono inserimenti e cancellazioni ma con il vincolo che gli inserimenti possono avvenire solo alla "fine" (rear) e le cancellazioni possono avvenire solo all' "inizio" (front) della coda

Quindi necessariamento il primo estratto è il primo inserito.

Non si può estrarre (dequeue) un elemento da una coda vuota, nè si può inserire (enqueue) un elemento in una pila piena.

La disciplina di accesso FIFO (First In First Out) è adatta a rappresentare sequenze nelle quali l'elemento viene elaborato secondo l'ordine di arrivo (lista d'attesa, insieme di dispositivi in attesa di assegnazione di risorse, etc.)

CODA: la specificazione

```
/* La specificazione della coda*/


typedef struct coda * CodaP;

CodaP  costrCoda( int max_num);
/* alloca la memoria per una nuova coda che può contenere fino a max_num
elementi.
 Prec: max_num >0
 postc: restituisce un puntatore alla nuova coda, esce se non c'è memoria */

void distrCoda(CodaP q);
/*prec:  q  è una coda costruita con costrCoda && q!=NULL
postc: libera la memoria impegnata dalla coda */

void  azzera(CodaP q);
/* azzera una coda consentendo di riutilizzarla senza riallocare la memoria
senza modificare q -> numMax.
 Prec: q  è una coda costruita con costrCoda && q!=NULL
postc: restituisce la coda svuotata*/
```

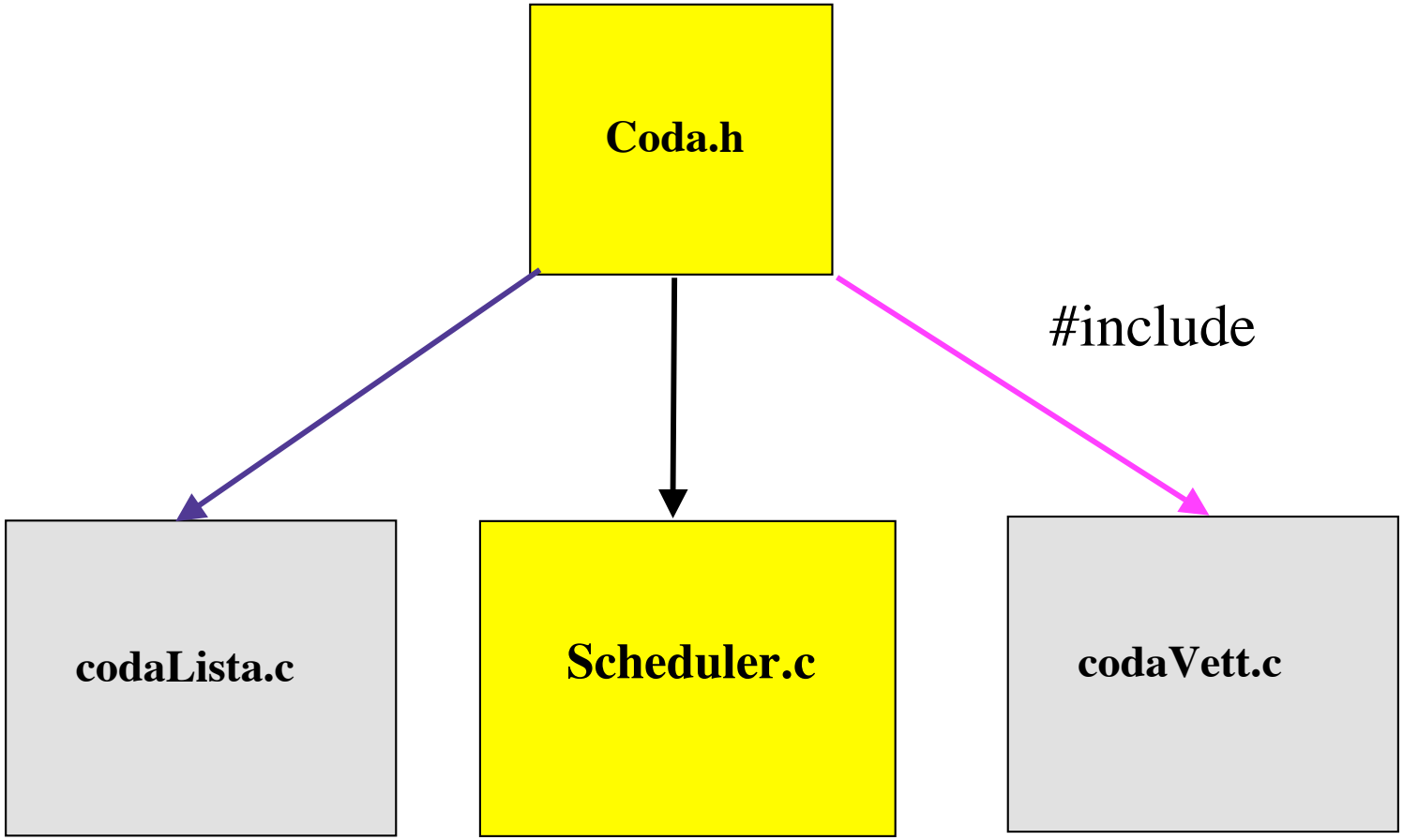
CODA: la specificazione

```
int  vuota(const CodaP q);
/* da' vero se la coda e' vuota, falso altrimenti
*prec: q  è una coda costruita con costrCoda && q!=NULL
postc: restituisce un valore !=0 se q è vuota, 0 altrimenti*/

int  piena(const CodaP q);
/* da' vero se la coda e' piena, falso altrimenti
*prec: q  è una coda costruita con costrCoda && q!=NULL
postc: restituisce un valore !=0 se q è piena, 0 altrimenti*/

int     primo(const CodaP q);
/* legge e restituisce il primo elemento in coda, se non e' vuoto
*prec: q  è una coda costruita con costrCoda && !vuota(q)
postc: restituisce il primo elemento in coda */
```

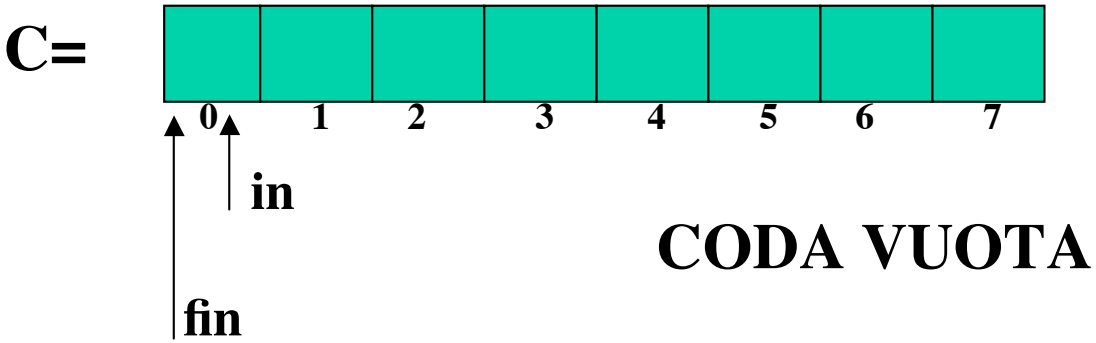
CODA: la specificazione

```
void  accoda(int el, CodaP q);
/* accoda el nella coda
prec: q  è una coda costruita con costrCoda && !piena(q)
postc: el è accodato in q*/

int  estrae(CodaP q);
/* restituisce ed elimina il primo elemento inserito nella coda, se non e' vuota
*prec: q  è una coda costruita con costrCoda &&  !vuota(q)
postc: restituisce il  primo elemento, eliminandolo dalla coda  */
```

CODA: esempio di uso. Una delle due implementazioni può essere utilizzata senza alcuna modifica!



CODA: implementazione su un vettore, versione 1



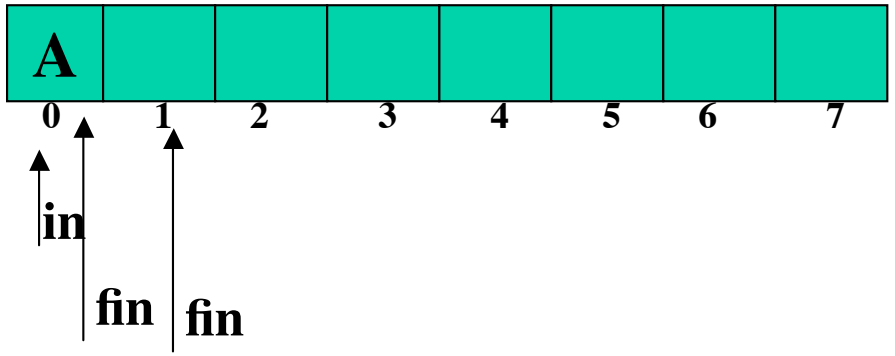
Gli elementi della coda sono quelli nel vettore C compresi tra C[in] e C[fin-1] inclusi. (invariante dell'implementazione)

si inserisce in C[fin]
e si incrementa fin di uno.

si estrae C[in] e si incrementa in di uno

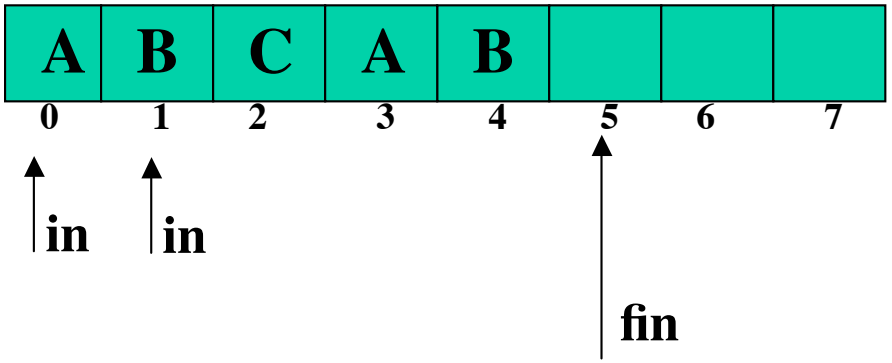
CODA: implementazione su un vettore, versione 1

un inserimento



dopo qualche inserimento

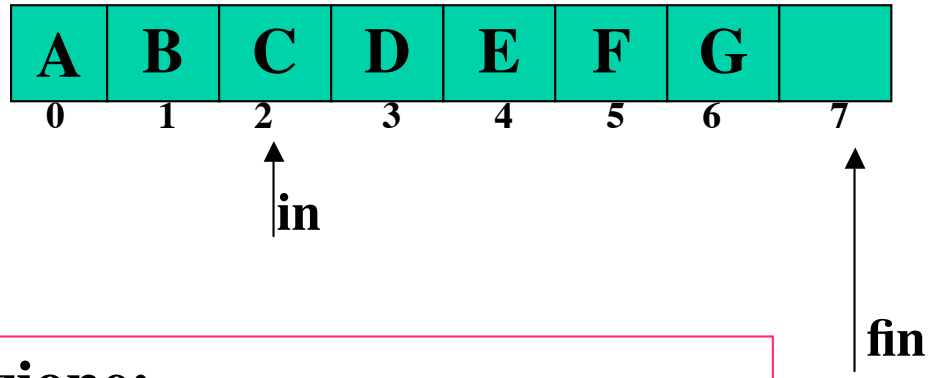
un' estrazione:



L'estrazione avviene salvando C[in] e incrementando in

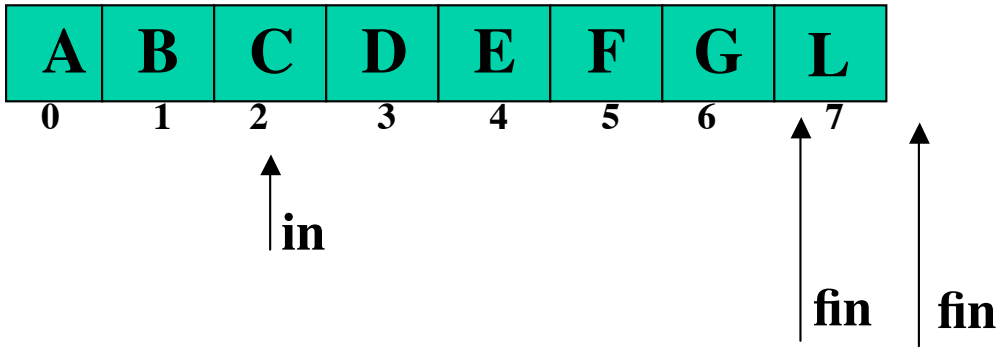
CODA: implementazione su un vettore, versione 1

Dopo vari inserimenti e qualche estrazione:



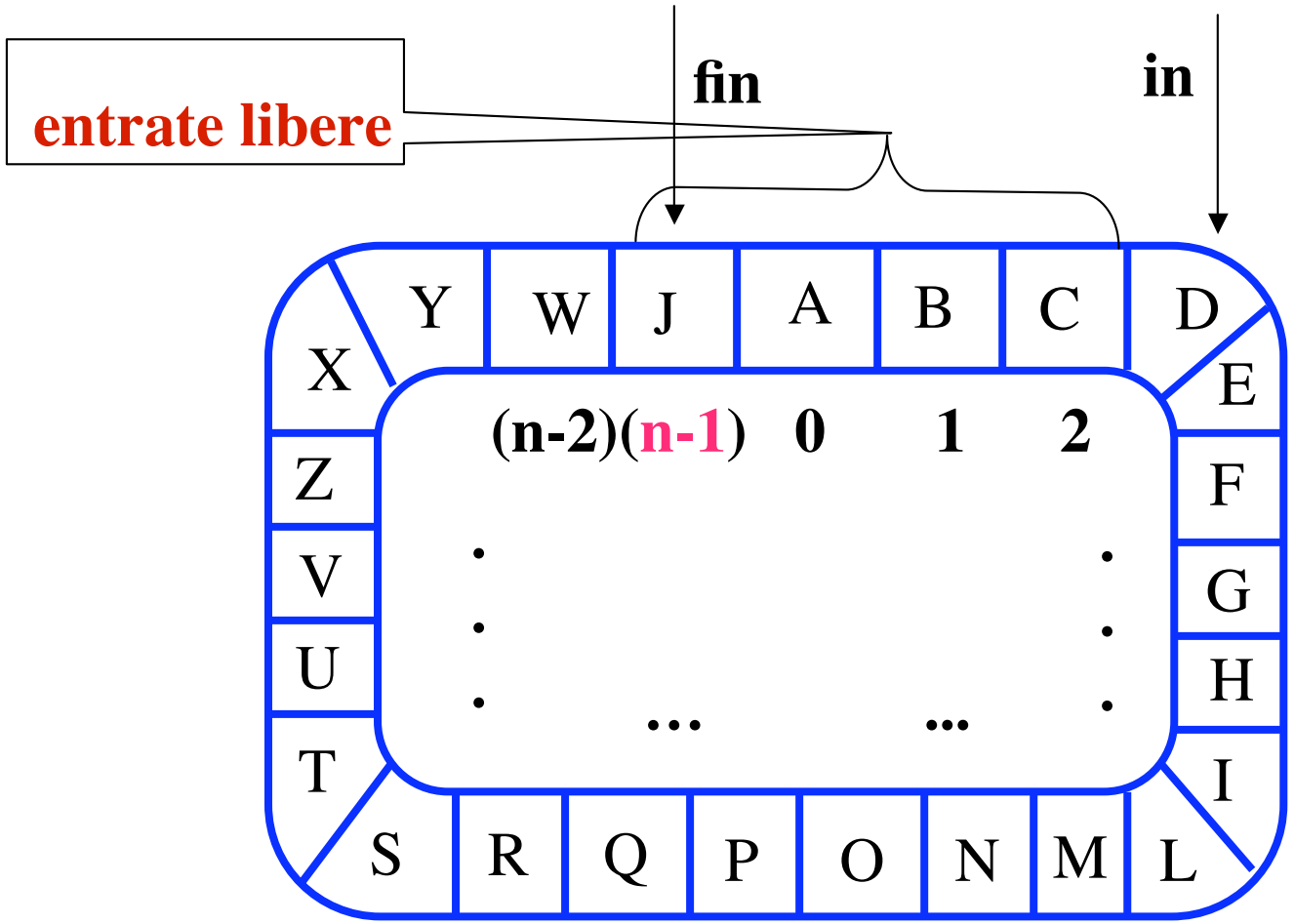
Invariante di implementazione:
gli elementi della coda sono quelli tra C[in] e C[fin-1]

Un nuovo inserimento:
si inserisce in C[fin]
e si incrementa fin di uno,
ma fin = n non è un valore ammesso!



L'aritmetica modulare è stata formalemente introdotta da Carl Friedrich Gauss nel 1801, nel suo *Disquisitiones Arithmeticae*.

L'osservazione base è che ogni intero n appartiene a una classe (di congruenza) di resti modulo m, cioè di resti della divisione di n per m.
Due numeri interi a e b sono congruenti modulo m, in breve a≡b(mod m) se la loro differenza (a−b) è un multiplo di m. Se 0≤b<m, a>0 e a ≡ b (mod m), allora b è il resto della divisione di a per m. Le classi dei resti mod m sono rappresentate da 0,1,…,m-1.
Se a≡b (mod m) allora a+c≡b+c (mod m) (invarianza rispetto alla somma).
In C abbiamo l'operatore a%m che fornisce il resto della divisione di a per m, per
Se 0<a<m-1 (a+1) %m=a+1 mentre se a = m-1 (a+1)%m = 0

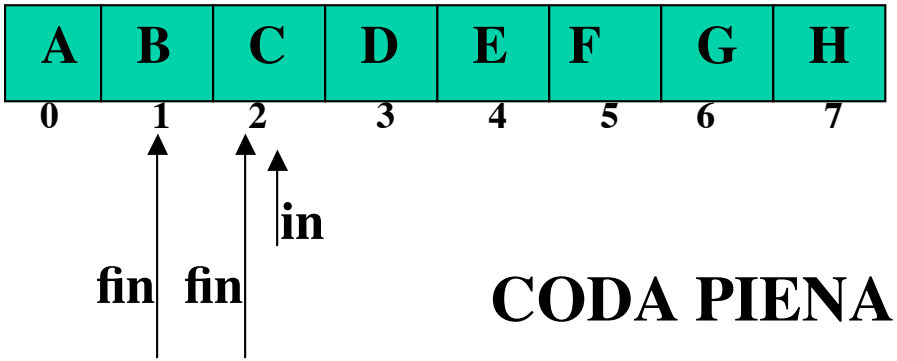


Gli elementi della coda sono quelli nel vettore C compresi tra C[in] e C[fin-1] inclusi.

L'incremento di fin dopo l'inserimento o di in dopo l'estrazione deve avvenire modulo n, la dimensione del vettore.

CODA: implementazione su un vettore, versione 1

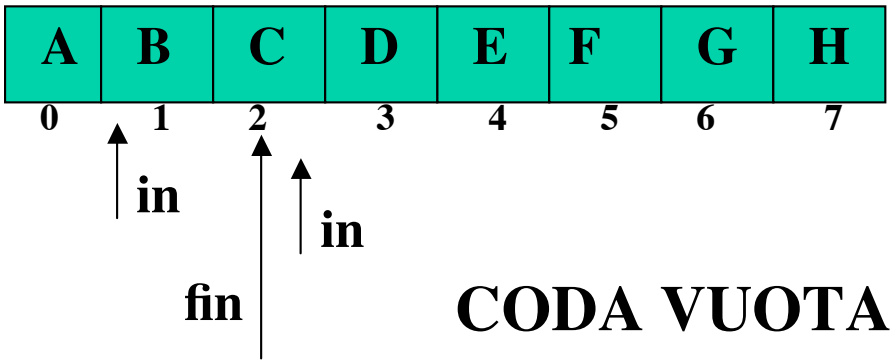
continuando ad inserire...:



CODA PIENA

inserimento in C[fin] e incremento di fin

continuando ad estrarre...:

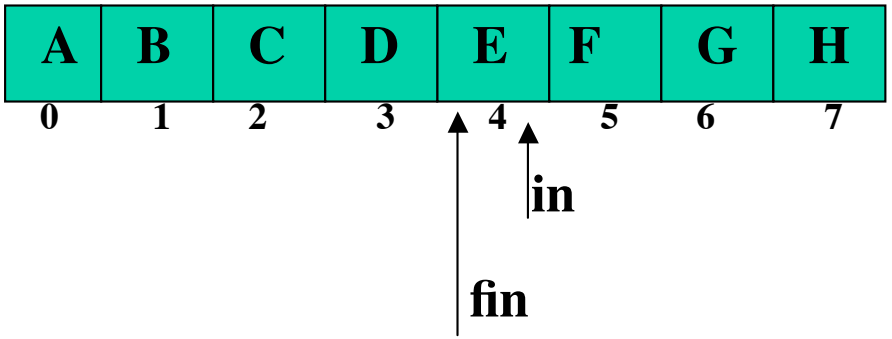


CODA VUOTA

estrazione di C[in] e incremento di in

CODA: implementazione su un vettore, versione 1

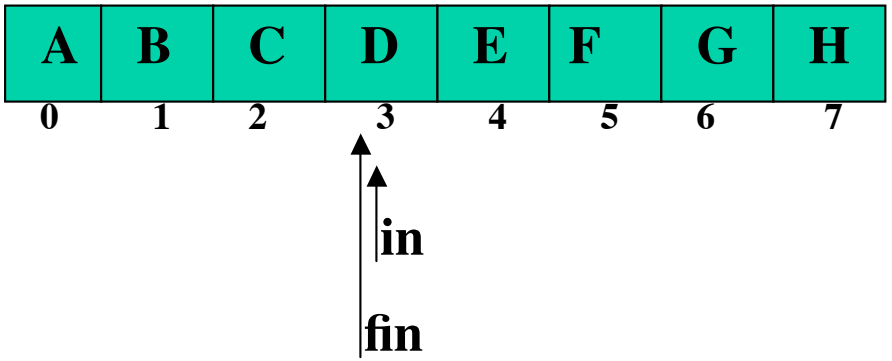
Introduciamo una variabile booleana, Vuota, inizialmente posta a TRUE e che ad ogni inserimento diventa FALSE



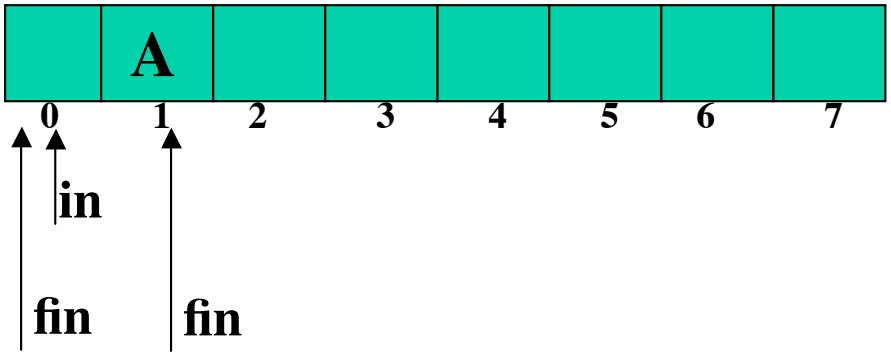
CODA VUOTA sse
in = fin e Vuota = TRUE

CODA PIENA sse
in = fin e Vuota = FALSE

Ad ogni estrazione, che può avvenire solo se Vuota è FALSE, si controlla se l'incremento di in porterà allo stesso valore di fin e in tal caso Vuota diventa di nuovo TRUE.



CODA: implementazione su un vettore, versione 2



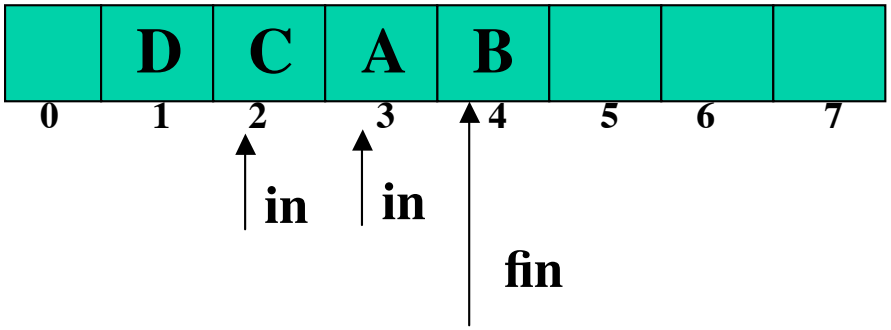
Nella **versione precedente** gli elementi della coda erano quelli compresi tra C[in] e C[fin-1] inclusi.

Sacrificando una entrata del vettore si evita di dover introdurre una nuova variabile e i relativi controlli.
Gli elementi della coda sono quelli nel vettore C compresi tra C[in+1] e C[fin] inclusi (invariante dell'implementazione).

Ora l'inserimento avviene incrementando fin, modulo la dimensione del vettore, e poi copiando in C[fin] il valore voluto.

CODA: implementazione su un vettore, versione 2

l'estrazione di A

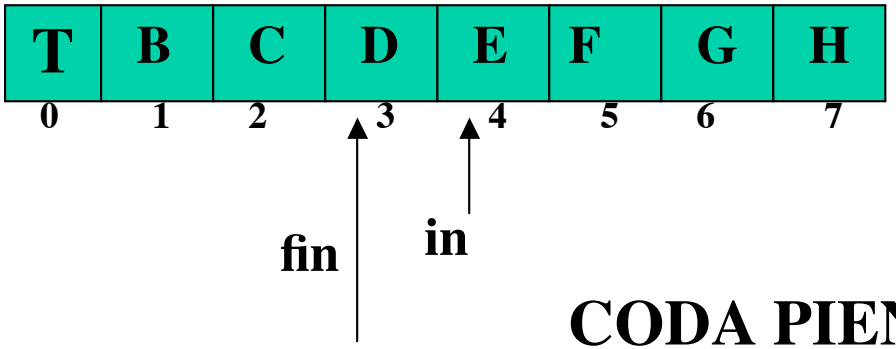


L'estrazione avviene salvando l'elemento C[in+1] e poi incrementando in, modulo la dimensione del vettore

Gli elementi della coda sono quelli nel vettore C compresi tra C[in+1] e C[fin] inclusi (invariante dell'implementazione).

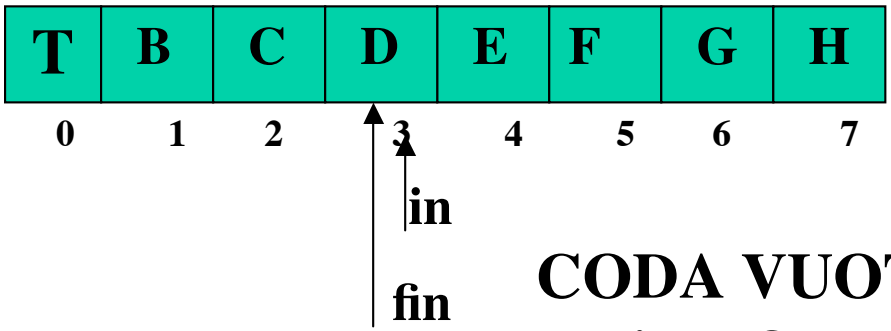
CODA: implementazione su un vettore, versione 2

L'inserimento avviene incrementando fin, modulo la dimensione del vettore, e poi copiando in C[fin] il valore voluto. Continuando a inserire, può accadere che fin+1 (modulo la dimensione del vettore) è uguale a in. Ma allora la coda è piena!



CODA PIENA
sse in = fin +1

L'estrazione avviene salvando l'elemento C[in+1] e poi incrementando in, modulo la dimensione del vettore Continuando a estrarre,
Può accadere che in (modulo la dimensione del vettore) è uguale a fin. Ma allora la coda è vuota!



CODA VUOTA
sse in = fin

CODA: implementazione su un vettore, versione 2. Il codice.

```
/* Nome file : codaVett.c
Implementazione della coda su un vettore di dimensione numMax. Invariante di
implementazione:
Dati due indici, inizio e fine, i cui valori sono compresi tra 0 e numMax-1, gli
elementi della coda  sono memorizzati nel vettore nelle posizioni  tra inizio+1 e fine
*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "Coda.h"


struct coda {
 int inizio;
 int fine;
 int numMax;
 int * elCoda;
};
```

CODA: implementazione su un vettore, versione 2. Il codice.

```
CodaP  costrCoda( int numMaxEl)
/* alloca la memoria per una nuova coda che può contenere fino a max_num elementi.
 Prec: numMaxEl >0
 postc: restituisce un puntatore alla nuova coda, esce se non c'è memoria */
{CodaP q;
assert(numMaxEl >0);
q = (CodaP) calloc(1,sizeof(struct coda));/*inizio=fine = numMax=0 elCoda = NULL*/
assert(!q);
q ->numMax = numMaxEl;
q->elCoda = malloc(numMaxEl*sizeof(int));
assert(!(q->elCoda));
return q;}

void distrCoda(CodaP q)
/*prec: q  è una coda costruita con costrCoda && q!=NULL
postc: libera la memoria impegnata dalla pila */
{assert(q);
 free(q -> elCoda);
free(q);
}
```

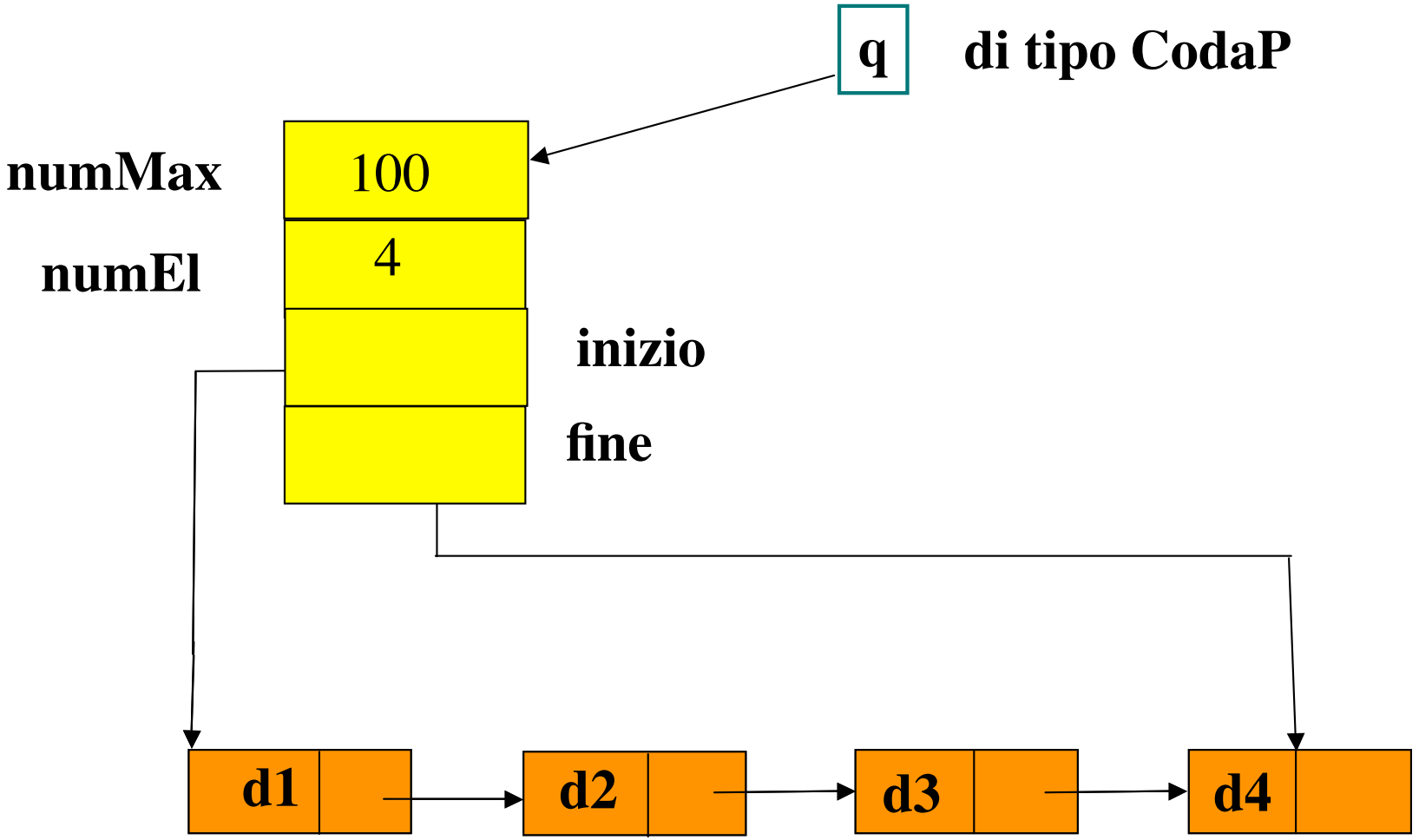
CODA: implementazione su un vettore, versione 2. Il codice.

```
void  azzera(CodaP q)
/* azzera una coda consentendo di riutilizzarla senza riallocare la memoria e
senza modificare q -> numMax.
 Prec: q  è una coda costruita con costrCoda && q!=NULL
 postc: restituisce la coda svuotata*/
{assert(q);
{q -> inizio = 0;
q -> fine = 0;}}

int vuota(const CodaP q)
/* da' vero se la coda e' vuota, falso altrimenti
*prec: q  è una coda costruita con costrCoda && q != NULL
postc: restituisce un valore !=0 se q è vuota, 0 altrimenti*/
 { assert(q);
   return (q->inizio == q->fine);}

int piena(const CodaP q)
/* da' vero se la coda e' piena, falso altrimenti
*prec: q  è una coda costruita con costrCoda && q != NULL
postc: restituisce unvalore !=0 se q è piena, 0 altrimenti*/
{assert(q);
 return (q->inizio == (q->fine + 1) % q->numMax);}
```

CODA: implementazione su un vettore, versione 2. Il codice.

```
int primo( CodaP q)
/* legge e restituisce il primo elemento in coda, se non e' vuoto
*prec: q  è una coda costruita con costrCoda && !vuota(q)
postc: restituisce il primo elemento in coda */
{assert(!vuota(q));
return q->elCoda[(q->inizio + 1) % q->numMax];}

void  accoda(int el, CodaP q)
/* accoda el nella coda
prec: q  è una coda costruita con costrCoda && !piena(q))
postc: el è accodato in q*/
{assert(!piena(q));
q->fine = (q->fine + 1) % q->numMax;
 q->elCoda[q->fine ] = el;}


int     estrae(CodaP q)
/* restituisce ed elimina il primo elemento inserito nella coda, se non e' vuota
*prec: q  è una coda costruita con costrCoda &&  !vuota(q)
postc: restituisce il  primo elemento, eliminandolo dalla coda  */
{assert(!vuota(q));
q->inizio = (q->inizio + 1) % q->numMax;
return q->elCoda[q->inizio];
}
```

CODA: implementazione su una lista.



CODA: implementazione su una lista. Il codice

```
/* Nome file: codaLista.c
coda implementata con una lista */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "Coda.h"

struct elem {              /* un elemento della coda */
  int  dato;
  struct elem   *next;
};

typedef   struct elem*   Elem;

struct coda {
   int    numEl;
   int    numMax;              /* un contatore per gli elementi*/
   Elem   inizio;            /* un puntatore al primo elemento*/
   Elem   fine;             /* un puntatore all'ultimo elemento*/
};
```

CODA: implementazione su una lista. Il codice

```
CodaP  costrCoda( int numMaxEl)
/* alloca la memoria per una nuova coda che può contenere fino a max_num elementi.
 Prec: numMaxEl >0
 postc: restituisce un puntatore alla nuova coda, NULL se non c'è memoria */
{CodaP q;
assert(numMaxEl >0);
q = (CodaP) calloc(1,sizeof(struct coda));/*inizio=fine = NULL, numMax=numEl=0 */
assert(!q);
q ->numMax = numMaxEl;
return q;}
```

CODA: implementazione su una lista. Il codice

```
void azzera(CodaP q)
/* azzera una coda consentendo di riutilizzarla senza riallocare la memoria
senza modificare q -> numMax.
 Prec: q  è una coda costruita con costrCoda && q!=NULL
 postc: restituisce la coda svuotata*/
{ Elem temp;
 assert(q != NULL);
   q -> numEl = 0;
   while (q->inizio)
              {temp = q->inizio;
              q -> inizio = q-> inizio-> next;
              free(temp);
              }
 q -> fine = NULL;
 }
```

CODA: implementazione su una lista. Il codice

```
void distrCoda(CodaP q)
/*prec: q  è una coda costruita con costrCoda && q!=NULL
postc: libera la memoria impegnata dalla pila */
{ Elem temp;
assert(q);
while (q->inizio)
            {temp = q->inizio;
            q -> inizio = q-> inizio-> next;
            free(temp);
            }
free(q);
}
```

CODA: implementazione su una lista. Il codice

```
int vuota( CodaP q)
/* da' vero se la coda e' vuota, falso altrimenti
*prec: q  è una coda costruita con costrCoda && q!=NULL
postc: restituisce unvalore !=0 se q è vuota, 0 altrimenti*/
{assert(q != NULL);
return ( (q -> numEl == 0));}

int piena( CodaP q)
/* da' vero se la coda e' piena, falso altrimenti
*prec: q  è una coda costruita con costrCoda && q != NULL
postc: restituisce unvalore !=0 se q è piena, 0 altrimenti*/
{assert(q != NULL);
return  q -> numEl == q -> numMax;}

int primo( CodaP q)
/* legge e restituisce il primo elemento in coda, se non e' vuoto
*prec: q  è una coda costruita con costrCoda && !vuota(q)
postc: restituisce il primo elemento in coda */
{assert(!vuota(q));
return (q -> inizio -> dato);}
```

CODA: implementazione su una lista. Il codice

```
int estrae(CodaP q)
/* restituisce ed elimina il primo elemento inserito nella coda, se non e' vuota
*prec: q  è una coda costruita con costrCoda &&  !vuota(q)
postc: restituisce il  primo elemento, eliminandolo dalla coda  */
{ int   d;
  Elem   p;
  assert(!vuota(q));
  d = q -> inizio -> dato;
  p = q -> inizio;
  q -> inizio = q -> inizio -> next;
  q -> numEl--;
  free(p);
  return d;}
```

CODA: implementazione su una lista. Il codice

```
void accoda(int el, CodaP q)
/* accoda el nella coda
prec: q  è una coda costruita con costrCoda &&  && !piena(q)
postc: el è accodato in q*/
{ Elem   new;
 assert(!piena(q));
 new = (Elem) calloc(1,sizeof(struct elem));
assert(!new);
new -> dato = el;
 if (!vuota(q))
              {q -> fine -> next =new;
               q -> fine = new;}
 else
               q -> fine = q -> inizio = new;
 q -> numEl++;
}
```

CODA: esempio di uso. Il codice

```
/*Nome del file: scheduler.c
Si usano due code per gestire nell'ordine di arrivo le richieste di risorse per un
sistema a   due processori*/

#include <stdio.h>
#include <assert.h>
#include "Coda.h"

#define MAX 100

int main(void)
{int max = MAX;
  int     c, cnt_a = 0, cnt_b = 0;        /*serviranno per numerare le richieste accodate*;
  unsigned int    pid;                        /* e' il numero identificativo del processo che
                                              richiede la risorsa */

  CodaP   a, b;
  a = costrCoda(max);
  b = costrCoda(max);
```

CODA: esempio di uso. Il codice

```
/*Fase di accodamento delle richieste */
  printf("\n scrivi A o B a seconda che sia una richiesta per il processore A o B,\
          C per terminare\n");
  while ((c = getchar()) != 'C')
         {switch (c)
              {case 'A':
                  printf("\n scrivi un numero che rappresenta una richiesta per il processore A\n");
                  assert(scanf("%u", &pid) == 1); /* inserisci i numeri  identificativi delle richieste */
                  if (!piena(a))
                  accoda(pid, a);
                   printf("\n scrivi A o B a seconda che sia una richiesta per il processore A o B,\
                                  C per terminare\n");
                  break;
             case 'B':
                 printf("\n scrivi un numero che rappresenta una richiesta per il processore A\n");
                                  C per terminare\n");
                assert(scanf("%u", &pid) == 1); /* inserisci i numeri  identificativi delle richieste */
                if (!piena(b))
                accoda(pid, b);
                printf("\n scrivi A o B a seconda che sia una richiesta per ilprocessore A o B,\
                                  C per terminare\n");
                break;
           }
      }
```

CODA: esempio di uso. Il codice

```
/*  Preleva dalla coda e stampa le richieste */
   printf(" il primo della coda a è %u\n",primo(a));
   printf(" il primo della coda b è %u\n",primo(b));
   printf("%s","\nElenco richieste processore A:\n");
   while (!vuota(a))
        { pid = estrae(a);
          printf("La richesta %u è %d\n", ++cnt_a, pid);
         }
   putchar('\n');
   printf("%s","\nElenco richieste processore B:\n");
   while (!vuota(b))
       { pid = estrae(b);
         printf("La richiesta %u è %d\n", ++cnt_b, pid);
       }
   putchar('\n');
   distrCoda(a);
   distrCoda(b);
   return 0;
}
```